USENIX Association

Third Computer Graphics Workshop

Monterey 1986

PROCEEDINGS

November 20-21, 1986

Monterey, California, USA

For additional copies of these proceedings, or copies of the Proceedings of the First or Second Computer Graphics Workshops, write

USENIX Association
P.O. Box 7
El Cerrito, CA 94530 USA

The price of the Third Proceedings is $10.00 per copy, plus $15 for overseas (air) postage.

The price per copy of the First and Second Proceedings is $3.00, plus $7 for overseas postage.

ACKNOWLEDGMENTS

Sponsored by:	USENIX Association P.O. Box 7 El Cerrito, CA 94530	
Program Chair:	Reidar Bornholdt	Columbia University
Program Committee:	Tom Duff Lou Katz Brian E. Redman	AT&T Bell Laboratories Metron Computerware, Ltd. Bell Communications Research
Workshop held at:	DoubleTree Hotel Monterey, California	
USENIX Meeting Planner:	Judith F. DesHarnais	
Proceedings Production:	Peter H. Salus Tom Strong	USENIX Executive Director Strong Consulting

TABLE OF CONTENTS

Opening Session

Chair: Lou Katz, Metron Computerware, Ltd.

Life Behind Screens

Chair: Brian E. Redman, Bell Communications Research

Making Pictures

Chair: Tom Duff, AT&T Bell Laboratories

Making Noise and Feeling Around

Chair: Michael Hawley, MIT Media Laboratory

The Utah Raster Toolkit

John W. Peterson
Rod G. Bogart
and
Spencer W. Thomas

University of Utah, Department of Computer Science
Salt Lake City, Utah

Abstract

The Utah Raster Toolkit is a set of programs for manipulating and composing raster images. These tools are based on the Unix concepts of pipes and filters, and operate on images in much the same way as the standard Unix tools operate on textual data. The Toolkit uses a special run length encoding (RLE) format for storing images and interfacing between the various programs. This reduces the disk space requirements for picture storage and provides a standard header containing descriptive information about an image. Some of the tools are able to work directly with the compressed picture data, increasing their efficiency. A library of C routines is provided for reading and writing the RLE image format, making the toolkit easy to extend.

This paper describes the individual tools, and gives several examples of their use and how they work together. Additional topics that arise in combining images, such as how to combine color table information from multiple sources, are also discussed.

1. Introduction

Over the past several years, the University of Utah Computer Graphics Lab has developed several tools and techniques for generating, manipulating and storing images. At first this was done in a somewhat haphazard manner - programs would generate and store images in various formats (often dependent on particular hardware) and data was often not easily interchanged between them. More recently, some effort has been made to standardize the image format, develop an organized set of tools for operating on the images, and develop a subroutine library to make extending this set of tools easier. This paper is about the result of this effort, which we call the *Utah Raster Toolkit.* Using a single efficient image format, the toolkit provides a number of programs for performing common operations on images. These are in turn based on a subroutine library for reading and writing the images.

2. Origins

The idea for the toolkit arose while we were developing an image compositor. The compositor (described in detail in section 4.1) allows images to be combined in various ways. We found that a number of simple and independent operations were frequently needed before images could be composited together.

With the common image format, the subroutine library for manipulating it, and the need for a

number of independent tools, the idea of a "toolkit" of image manipulating programs arose. These tools are combined using the Unix shell, operating on images much like the standard Unix programs operate on textual data. For example, a Unix user would probably use the sequence of commands:

```
cat /etc/passwd | grep Smith | sort -t: +4.0 | lpr
```

to print a sorted list of all users named "Smith" in the system password file. Similarly, the Raster Toolkit user might do something like:

```
cat image.rle | avg4 | repos -p 0 200 | getfb
```

to downfilter an image and place it on top of the frame buffer screen. The idea is similar to a method developed by Duff [3] for three dimensional rendering.

3. The RLE format

The basis of all of these tools is a Run Length Encoded image format [8]. This format is designed to provide an efficient, device independent means of storing multi-level raster images. It is not designed for binary (bitmap) images. It is built on several basic concepts. The central concept is the *channel*. A channel corresponds to a single color, thus there are normally separate red, green and blue channels. Up to 255 color channels are available for use; one channel is reserved for coverage ("alpha") data. Although the format supports arbitrarily deep channels, the current implementation is restricted to 8 bits per channel. An RLE file is treated as a byte stream, making it independent of host byte ordering.

Image data is stored in an RLE file in a scanline form, with the data for each channel of the scanline grouped together. Runs of identical pixel values are compressed into a count and a value. However, sequences of differing pixels are also stored efficiently (i.e, not as a sequence of single pixel runs).

3.1. The RLE header

The file header contains a large amount of information about the image. This includes:

- The size and position on the screen,
- The number of channels saved and the number of bits per channel (currently, only eight bits per channel is supported),
- Several flags, indicating: how the background should be handled, whether or not an alpha channel was saved, if picture comments were saved,
- The size and number of channels in the color map, (if the color map is supplied),
- An optional background color,
- An optional color map,
- An optional set of comments. The comment block contains any number of null-terminated text strings. These strings are conventionally of the form "name=value", allowing for easy retrieval of specific information.

3.2. The scanline data

The scanline is the basic unit that programs read and write. It consists of a sequence of operations, such as *Run*, *SetChannel*, and *Pixels*, describing the actual image. An image is stored starting at the lower left corner and proceeding upwards in order of increasing scanline number. Each operation and its associated data takes up an even number of bytes, so that all operations begin on a 16 bit boundary. This makes the implementation more efficient on many architectures.

Each operation is identified by an 8 bit opcode, and may have one or more operands. Single operand operations fit into a single 16 bit word if the operand value is less than 256. So that operand values are not limited to the range 0..255, each operation has a *long* variant, in which the

byte following the opcode is ignored and the following word is taken as a 16 bit quantity. The long variant of an opcode is indicated by setting the bit 0x40 in the opcode (this allows for 64 opcodes, of which 6 have been used so far.)

The current set of opcodes include:

SkipLines	Increment the *scanline number* by the operand value, terminating the current scanline.
SetColor	Set the *current channel* to the operand value.
SkipPixels	Skip over pixels in the current scanline. Pixels skipped will be left in the background color.
PixelData	Following this opcode is a sequence of pixel values. The length of the sequence is given by the operand value.
Run	This is the only two operand opcode. The first operand is the length (*N*) of the run. The second operand is the pixel value, followed by a filler byte if necessary[1]. The next *N* pixels in the scanline are set to the given pixel value.
EOF	This opcode has no operand, and indicates the end of the RLE file. It is provided so RLE files may be concatenated together and still be correctly interpreted. It is not required, a physical end of file also indicates the end of the RLE data.

3.3. Subroutine Interface

Two similar subroutine interfaces are provided for reading and writing files in the RLE format. Both read or write a scanline worth of data at a time. A simple "row" interface communicates in terms of arrays of pixel values. It is simple to use, but slower than the "raw" interface, which uses arrays of "opcode" values as its communication medium.

In both cases, the interface must be initialized by calling a setup function. The two types of calls may be interleaved; for example, in a rendering program, the background could be written using the "raw" interface, while scanlines containing image data could be converted with the "row" interface. The package allows multiple RLE streams to be open simultaneously, as is necessary for use in a compositing tool, for example. All data relevant to a particular RLE stream is contained in a "globals" structure. This structure essentially echoes the information in the RLE header, along with current state information about the RLE stream.

4. The tools

4.1. The image compositor - Comp

Comp implements an image compositor based on the compositing algorithms presented in [6]. The compositing operations are based on the presence of an alpha channel in the image. This extra channel usually defines a mask which represents a sort of a cookie-cutter for the image. This is the case when alpha is 255 (full coverage) for pixels inside the shape, zero outside, and between zero and 255 on the boundary. When the compositor operates on images of the cookie-cutter style, the operations behave as follows:

A **over** B (the default)	The result will be the union of the two image shapes, with A obscuring B in the region of overlap.
A **atop** B	The result shape is the same as image B, with A obscuring B where the

[1] E.g., a 16 bit pixel value would not need a filler byte.

image shapes overlap. (Note that this differs from "over" because the portion of A outside the shape of B will not be in the result image.)

A **in** B
: The result is simply the image A cut by the shape of B. None of the image data of B will be in the result.

A **out** B
: The result image is image A with the shape of B cut out.

A **xor** B
: The result is the image data from both images that is outside the overlap region. The overlap region will be blank.

A **plus** B
: The result is just the sum of the image data. This operation is actually independent of the alpha channels.

The alpha channel can also represent a semi-transparent mask for the image. It would be similar to the cookie-cutter mask, except the interior of the shape would have alpha values that represent partial coverage; e.g. 128 is half coverage. When one of the images to be composited is a semi-transparent mask, the following operations have useful results:

Semi-transparent A **over** B
: The image data of B will be blended with that of A in the semi-transparent overlap region. The resulting alpha channel is as transparent as that of image B.

A **in** Semi-transparent B
: The image data of A is scaled by the mask of B in the overlap region. The alpha channel is the same as the semi-transparent mask of B.

If the picture to be composited doesn't have an alpha channel present, comp assumes an alpha of 255 (i.e., full coverage) for non-background pixels and an alpha of zero for background pixels.

Comp is able to take advantage of the size information for the two images being composited. For example, if a small picture is being composited over a large backdrop, the actual compositing arithmetic is only performed on a small portion of the image. By looking at the image size in the RLE header, comp performs the compositing operation only where the images overlap. For the rest of the image the backdrop is copied (or not copied, depending on the compositing operation). The "raw" RLE read and write routines are used to perform this copy operation, thus avoiding the cost of compressing and expanding the backdrop as well.

4.2. Basic image composition - Repos and Crop

`Repos` and `crop` are the basic tools for positioning and arranging images to be fed to the compositor. Crop simply throws away all parts of the image falling outside the rectangle specified. Repos positions an image to a specific location on the screen, or moves it by an incremental amount. Repos does not have to modify any of the image data, it simply changes the position specification in the RLE header. In order to simplify the code, the Raster Toolkit does not allow negative pixel coordinates in images.

4.3. Changing image orientation and size - Flip, Fant and Avg4

Two tools exist for changing the orientation of an image on the screen. `Flip` rotates an image by 90 degrees right or left, turns an image upside down, or reverse it from right to left. `Fant` rotates an image an arbitrary number of degrees from -45 to 45. In order to get rotation beyond -45 to 45, flip and fant can be combined, for example:

```
flip -r < upright.rle | fant -a 10 > rotated.rle
```

rotates an image 100 degrees. Fant is also able to scale images by an arbitrary amount in X and Y. A common use for this is to stretch or shrink an image to correct for the aspect ratio of a particular frame buffer. Many frame buffers designed for use with standard video hardware display a picture of 512x480 pixels on a screen with the proportions 4:3, resulting in an overall pixel aspect ratio of 6:5. If the picture `kloo.rle` is digitized or computed assuming a 1:1 aspect ratio (square pixels), the command:

```
cat kloo.rle | fant -s 1.0 1.2 | getfb
```

correctly displays the image on a frame buffer with a 6:5 aspect ratio. Fant is implemented using a two-pass subpixel sampling algorithm [4]. This algorithm performs the spatial transform (rotate and/or scale) first on row by row basis, then on a column by column basis. Because the transformation is done on a subpixel level, fant does not introduce aliasing artifacts into the image.

Avg4 downfilters an RLE image into a resulting image of 1/4th the size, by simply averaging four pixel values in the input image to produce a single pixel in the output. If the original image does not contain an alpha channel, avg4 creates one by counting the number of non-zero pixels in each group of four input pixels and using the count to produce a coverage value. While the alpha channel produced this way is crude (only four levels of coverage) it is enough to make a noticeable improvement in the edges of composited images. One use for avg4 is to provide anti-aliasing for rendering programs that perform no anti-aliasing of their own. For example, suppose **huge.rle** is a 4k x 4k pixel image rendered without anti aliasing and without an alpha channel. Executing the commands:

```
cat huge.rle | avg4 | avg4 | avg4 > small.rle
```

produces an image **small.rle** with 64 (8x8) samples per pixel and an alpha channel with smooth edges.

Images generated from this approach are as good as those produced by direct anti-aliasing algorithms such as the A-buffer [2]. However, a properly implemented A-buffer renderer produces images nearly an order of magnitude faster.

4.4. Color map manipulation - Ldmap and Applymap

As mentioned previously, RLE files may optionally contain a color map for the image. **Ldmap** is used to create or modify a color map within an RLE file. Ldmap is able to create some standard color maps, such as linear maps with various ramps, or maps with various gamma corrections. Color maps may also be read from a simple text file format, or taken from other RLE files. Ldmap also performs *map composition*, where one color map is used as in index into the other. An example use for this is to apply a gamma correction to an image already having a non-linear map.

Applymap applies the color map in an rle file to the pixel values in the file. For example, if the color map in **kloo.rle** contained a color map with the entries (for the red channel):

```
Index   Red color map
    0:      5
    1:      7
    2:      9
```

Then a (red channel) pixel value of zero would be displayed with an intensity of five (assuming the display program used the color map in **kloo.rle**). When kloo.rle is passed through applymap,

```
cat kloo.rle | applymap > kloo2.rle
```

pixels that had a value of zero in **kloo.rle** now have a value of five in **kloo2.rle**, pixel values of one would now be seven, etc. When displaying the images on a frame buffer, **kloo2.rle** appears the same with a linear map loaded as **kloo.rle** does with its special color map loaded.

One use for these tools is merging images with different compensation tables. For example, suppose image **gam.rle** was computed so that it requires a gamma corrected color table (stored in the RLE file) to be loaded to look correct on a particular monitor, and image **lin.rle** was computed with the gamma correction already taken into consideration (so it looks correct with a linear color table loaded). If these two images are composited together without taking into consideration these differences, the results aren't correct (part of the resulting image is either too dim or washed out). However, we can use applymap to "normalize" the two images to using a linear map with:

```
cat gam.rle | applymap | comp - lin.rle | ldmap -l > result.rle
```

4.5. Generating backgrounds

Unlike most of the toolkit programs which act as filters, **background** just produces output. Background either produces a simple flat field, or it produces a field with pixel intensities ramped in the vertical direction. Most often background is used simply to provide a colored backdrop for an image. Background is another example of a program that takes advantage of the "raw" RLE facilities. Rather than generating the scanlines and compressing them, background simply generates the opcodes required to produce each scanline.

4.6. Converting full RGB images to eight bits - to8 and tobw

Although most of the time we prefer to work with full 24 bit per pixel images, (32 bits with alpha) we often need the images represented with eight bits per pixel. Many inexpensive frame buffers (such as the AED 512) and many personal color workstations (VaxStation GPX, Color Apollos and Suns) are equipped with eight bit displays.

To8 converts a 24 bit image to an eight bit image by applying a dither matrix to the pixels. This basically trades spatial resolution for color resolution. The resulting image has eight bits of data per pixel, and also contains a special color map for displaying the dithered image (since most eight bit frame buffers still use 24 bit wide color table entries).

Tobw converts a picture from 24 bits of red, green, and blue to an eight bit gray level image. It uses the standard television YIQ transformation of

$$graylevel = 0.35 \times red + 0.55 \times green + 0.10 \times blue.$$

These images are often preferred when displaying the image on an eight bit frame buffer and shading information is more important than color.

5. Interfacing to the RLE toolkit

In order to display RLE images, a number of programs are provided for displaying the pictures on various devices. These display programs all read from standard input, so they are conveniently used as the end of a raster toolkit pipeline. The original display program, **getfb**, displays images on our ancient Grinnell GMR-27 frame buffer. Many display programs have since been developed:

getcx	displays images on a Chromatics CX1500
getiris	displays an image on an Iris workstation (via ethernet)
getX	for the X window system
getap	for the Apollo display manager
gethp	for the Hewlett-Packard Series 300 workstation color display
rletops	converts an RLE file to gray-level PostScript output[2]

The **getX**, **getap** and **gethp** programs automatically perform the dithering required to convert a 24 bit RLE image to eight bits (for color nodes) or one bit (for bitmapped workstations). Since all of these workstations have high resolution (1Kx1K) displays, the trade of spatial resolution for color resolution produces very acceptable results. Even on bitmapped displays the image quality is good enough to get a reasonable idea of an image's appearance.

Two programs, **painttorle** and **rletopaint** are supplied to convert MacPaint images to RLE files. This offers a simple way to add text or graphic annotation to an RLE image (although the resolution is somewhat low).

[2] **Rletops** was used to produce the figures in this paper.

6. Examples

A typical use for the toolkit is to take an image generated with a rendering program, add a background to it, and display the result on a frame buffer:

```
rlebg 250 250 250 -v | comp image.rle - | getfb
```

What follows are some more elaborate applications:

6.1. Making fake shadows

In this exercise we take the dart (Figure 6-1a) and stick it into the infamous mandrill, making a nice (but completely fake) shadow along the way.

Figure 6-1: **a:** Original dart image. **b:** Rotated and stretched dart.

First the image of the dart is rotated by 90 degrees (using `rleflip -l` and then stretched in the X direction (using `fant -s 1.3 1.0`) to another image we can use as a shadow template, figure 6-1b. Then we take this image, `dart_stretch_rot.rle`, and do:

```
rlebg 0 0 0 175 \
    | comp -o in - dart_stretch_rot.rle > dart_shadow.rle
```

This operation uses the stretched dart as a cookie-cutter, and the shape is cut out of a black image, with a coverage mask (alpha channel) of 175. Thus when we composite the shadow (figure 6-2a) over the mandrill image, it lets 32% of the mandrill through ((255 - 175) / 255 = 32%) with the rest being black.

Now all we have left to do is to composite the dart and the shadow over the mandrill image, and the result is figure 6-2b. Note that since the original dart image was properly anti-aliased, no aliasing artifacts were introduced in the final image.

6.2. Cutting holes

In this example we start out out with a single bullet hole, created with the same modelling package that produced the dart.[3] First the hole (which originally filled the screen) is downfiltered to

[3]The bullet hole was modeled with cubic B-splines. Normal people probably would have painted something like this...

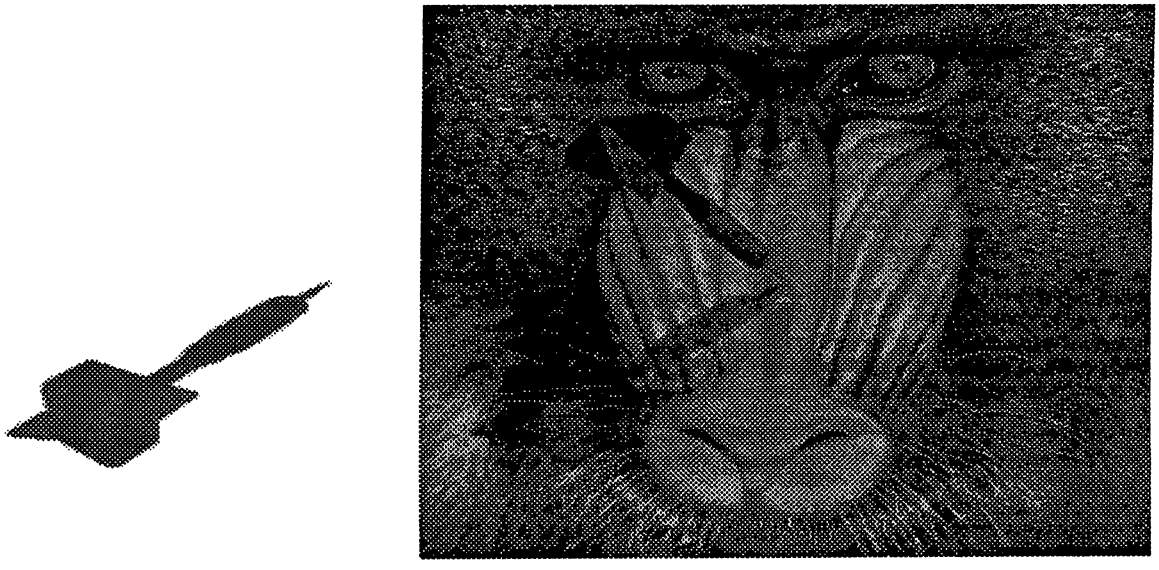


Figure 6-2: **a:** Dart shadow mask. **b:** Resulting skewered baboon.

a small size, with

```
avg4 < big_hole.rle | avg4 | avg4 > smallhole.rle
```

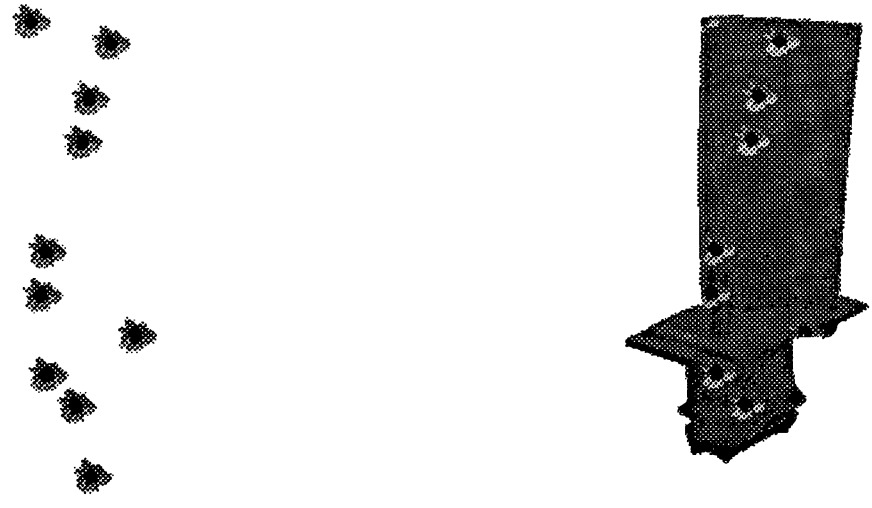


Figure 6-3: **a:** A set of bullet holes. **b:** A shot-up turbine blade.

Now by invoking `repos` and `comp` several times in a row, a set of shots is built up (Figure 6-3a)

To shoot up the turbine blade, the *atop* operator of comp is used. This works much like *over*, except the bullet holes outside of the turbine blade don't appear in the resulting image (Figure 6-3b) Note how the top left corner is just grazed.

But there's still one catch. Suppose we want to display the shot-up turbine blade over a background of some sort. We want to be see the background through the holes. To do this we come up with another mask to cut away the centers of the bullet holes already on the turbine blade so we

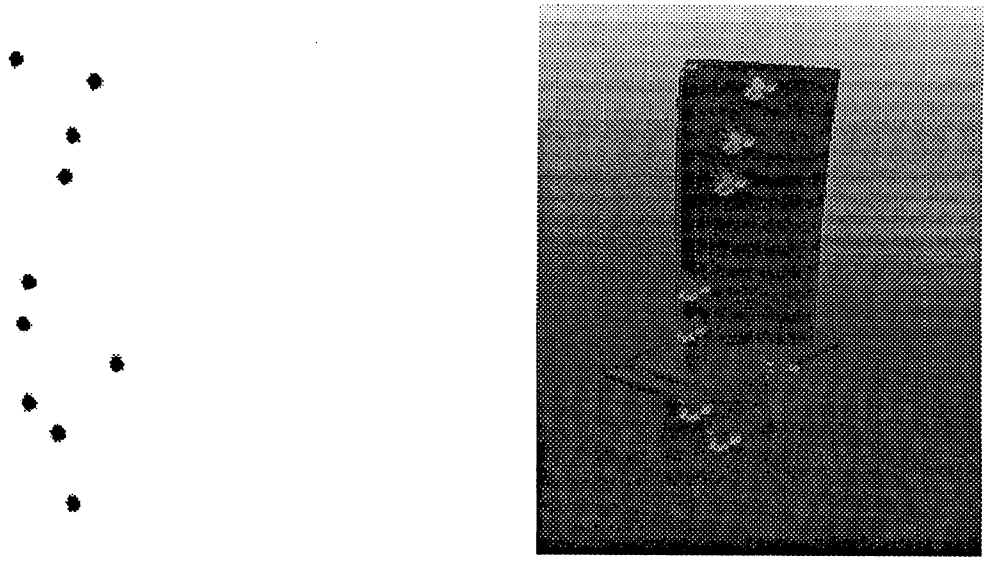


Figure 6-4: **a:** Cut-away masks for the bullet hole centers.
b: Finished turbine blade

can see through them (Figure 6-4a).Now we can see through the blade (Figure 6-4b).

6.3. Integrating digitized data

In this example, we take a raw digitized negative and turn it into a useful frame buffer image. Figure 6-5a shows the original image from the scanner. First the image is cropped to remove the excess digitized portion, then rotated into place with `flip -r`, resulting in Figure 6-5b.

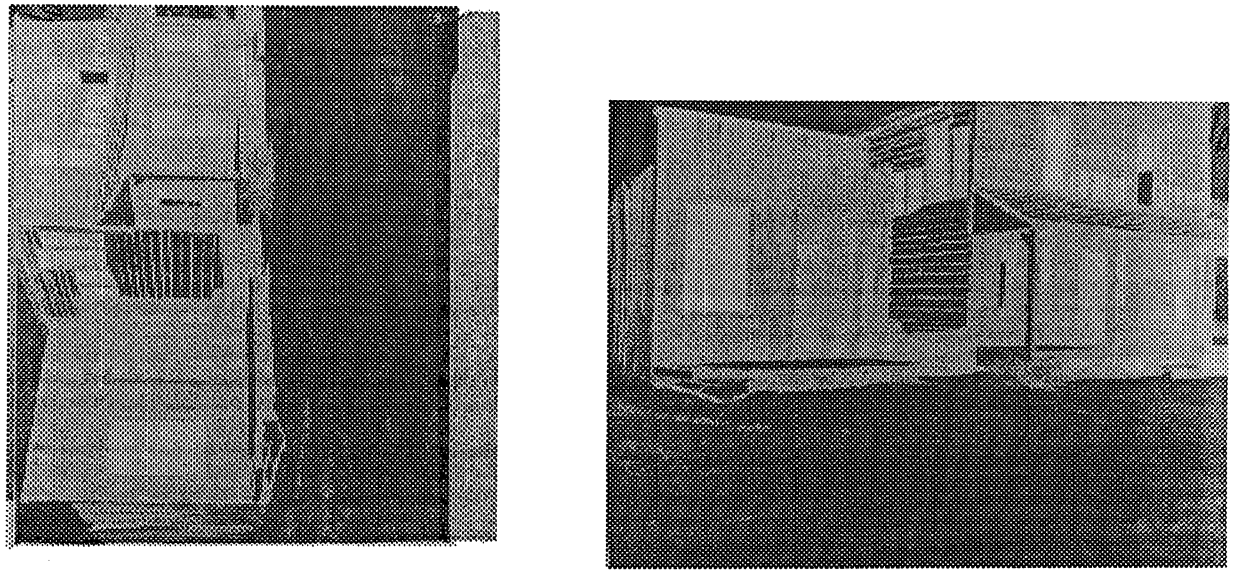


Figure 6-5: **a:** The raw digitized negative. **b:** Cropped and rotated image

To get the image to appear as a positive on the frame buffer, a negative linear color map (stored as text file) is loaded, then this is composed with a gamma correction map of 2.2, and finally applied to the pixels with the command:

```
ldmap -f neg.cmap < neg_pahriah.rle | ldmap -a -g 2.2 \
   | applymap > pahriah.rle
```

The result, Figure 6-6 now appears correct with a linear color map loaded.

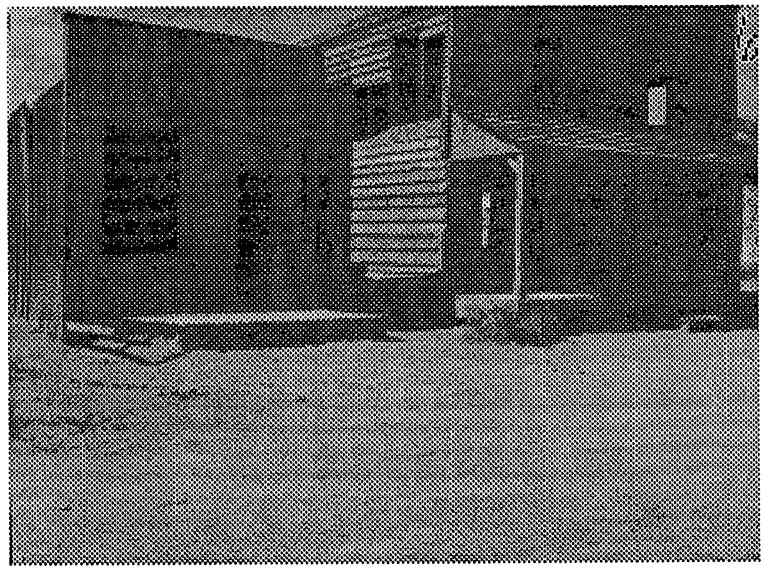


Figure 6-6: Final image of the old Pahriah ghost town

7. Future Work

Needless to say, a project such as the Raster Toolkit is open ended in nature, and new tools are easily added. For example, most of the tools we have developed to date deal with image synthesis and composition, primarily because that is our research orientation. Additional tools could be added to assist work in areas such as vision research or image processing (for examples, see [1, 7]).

Another interesting application would be a visual "shell" for invoking the tools. Currently, arguments to programs like `crop` and `repos` are specified by tediously finding the numbers with the frame buffer cursor and then typing them into a shell. The visual shell would allow the various toolkit operations to be interactively selected from a menu. Such a shell could probably work directly with an existing workstation window system such as X [5] to provide a friendly environment for using the toolkit.

8. Conclusions

The Raster Toolkit provides a flexible, simple and easily extended set of tools for anybody working with images in a Unix-based environment. We have ported the toolkit to a wide variety of Unix systems, including Gould UTX, HP-UX, Sun Unix, Apollo Domain/IX, 4.2 and 4.3 BSD, etc. The common interfaces of the RLE format and the subroutine library make it easy to interface the toolkit to a wide variety of image sources and displays.

9. Acknowledgments

This work was supported in part by the National Science Foundation (DCR-8203692 and DCR-8121750), the Defense Advanced Research Projects Agency (DAAK11-84-K-0017), the Army Research Office (DAAG29-81-K-0111), and the Office of Naval Research (N00014-82-K-0351). All opinions, findings, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Burt, Peter J, and Adelson, Edward H.
A Multiresolution Spline With Application to Image Mosaics.
ACM Transactions on Graphics 2(4):217-236, October, 1983.

[2] Carpenter, Loren.
The A-Buffer, An Antialiased Hidden Surface Method.
Computer Graphics 18(3):103, July, 1984.
Proceedings of SIGGRAPH 84.

[3] Duff, Tom.
Compositing 3-D Rendered Images.
Computer Graphics 19(3):41, July, 1985.
Proceedings of SIGGRAPH 85.

[4] Fant, Karl M.
A Nonaliasing, Real-Time, Spatial Transform.
IEEE Computer Graphics and Applications 6(1):71, January, 1986.

[5] Gettys, Jim, Newman, Ron, and Fera, Tony D.
Xlib - C Language X Interface, Protocol Version 10.
Technical Report, MIT Project Athena, January, 1986.

[6] Porter, Thomas and Duff, Tom.
Compositing Digital Images.
Computer Graphics 18(3):253, July, 1984.
Proceedings of SIGGRAPH 84.

[7] Stockham, Thomas G.
Image Processing in the Context of a Visual Model.
Proceedings of the IEEE 60(7):828-842, July, 1972.

[8] Thomas, Spencer W.
Design of the Utah RLE Format.
Technical Report 86-15, Alpha_1 Project, CS Department, University of Utah, November, 1986.

A High-End High-Performance Graphics System for Computational Fluid Dynamics†

Julian E. Gomez
Research Institute for Advanced Computer Science

Frank Preston
National Aeronautics and Space Administration

Steve Fine
Tony Hasegawa
Bock Lee
Blaine Walker
General Electric Western Systems, Space Systems Division

Computational Fluid Dynamics (CFD) presents interesting problems for the design of a graphics system in that it requires near real time performance for very large databases. The Numerical Aerodynamic Simulation (NAS) program at the Ames Research Center is responsible for building support systems for CFD activities. The graphics project at NAS was organized to design and build the high performance graphics portion.

A basic requirement for a new graphics system is that it be able to support current graphics activity. Otherwise scientists are thrown from one operating paradigm to another, a technique which is generally counterproductive. Furthermore, a new system should demonstrate performance greater than the system it's supplementing.

Just as important, however, is that the new system provide advanced and flexible methods of computer graphics to support leading edge CFD activity. This includes, for example, transparency, antialiasing and bicubic surfaces. Image processing functions will enhance pictures the scientist has drawn without resorting to a central computing site. Animation capabilities will aid the scientist in selecting parameter values to study. Commercial products are preferable, since Ames doesn't have the resources to design and build high-performance graphics equipment.

An interesting aspect of CFD graphics is that displays consist of 3-D volumes, not surfaces, *i.e.* it is at least as important to view the interior as it is to view the exterior. This makes variable levels of transparency essential to the display process, with interactive control being the most efficient way to study the data.

No matter what equipment is used for NAS advanced graphics, the software running on it will constantly be changing. Ames is a research center, and its activities represent experimentation. A program that was useful one day may require modification the next for an altered version of the experiment. The system thus needs a software development environment to facilitate the constantly changing state of software.

The high performance requirements are manifested in two ways. The first is a desire to download frames at a high rate (800Mbits/sec) from a high speed processor (HSP), such as a Cray 2, to the display device. In effect, the scientist would be able to watch the simulation in progress. Additionally, the system must be able to support a distributed

† The work described here was sponsored by the NAS program at the NASA Ames Research Center, Moffett Field, CA.

processing mode, where the scientist specifies some action using the graphics device, which then communicates to other processors to actually have the work performed.

Finally, the graphics system must be able to operate in a standalone fashion. Besides the fact that the HSP will not always be available, it is an expensive resource, and if the scientist is able to perform advanced functions on the graphics system, he will be able to avoid the cost and delays involved in using the Cray. Consequently, the system must be able to store precomputed images and play them back quickly. It must also be able to deal with graphics at a more abstract level, such as a collection of bicubic patches, so the scientist can redraw pictures of 3D surfaces and bodies without using the HSP. The scientist will be able to explore his CFD graphics locally but still use the Cray when necessary.

These requirements lead to providing the scientist with a collection of graphics capabilities collected in a convenient spot; along the lines of the *Programmer's Work Bench* we call this the *CFD Graphics Work Bench*. The work bench includes implementations of current NAS software as well as systems implementing the capabilities described above. There will also be tools to handle various activities such as supervising the distributed processing activities or designing a (software) control panel for a simulation. The approach of the workbench is to provide the scientist with equipment and software that will speed the process of graphical interpretation of large CFD problems as well as providing a testbed for CFD experimentation.

Recollections

Ed Tannenbaum
Eddeo Inc.
PO Box 92
Crockett, CA 94525

Recollections is an interactive video installation that was originally created during an Artist-in-Residency at the Exploratorium in San Francisco. I conceived the piece while a student at the Rhode Island School of Design in 1976, but it wasn't completed until June of 1981.

Recollections is the result of many influences. My father is a printer, so from early on I was exposed to elements of the graphic arts and remember being fascinated by the array of dots that create a halftone image in print and the way in which, at a certain point, the dots fuse together to form a recognizable picture from a seemingly abstract pattern. I appreciated the moment when the brain suddenly made sense out of chaos, and could practically feel the changeover as new perceptual connections and pathways were forged.

There are, of course, precedents throughout the history of art that have exploited this effect, the best known being Pointillism, a movement begun by the painter Georges Seurat in the late 19th century. In the 1880s, Etienne-Jules Marey, a French photographer, began experimenting with film as a means of analyzing movement. Unlike his contemporary Muybridge, Marey made multiple exposures on a single frame. His subjects included almost anything that moved -- from birds in flight to a bouncing ball to a nude descending a staircase. Technological innovations of the day have always influenced the vision of artists, sometimes suggesting new ways to see; Marcel Duchamp was inspired by Marey's photos and, in 1912, painted his masterpiece, *Nude Descending a Staircase*, one of the earliest works of analytical cubism.

Marey's work is, in itself, exemplary of the ways in which a simple, recognizable event can be captured through a direct photographic process and converted into a compelling abstract graphic. Our interest is invariably augmented by the fact that the images reveal the process of movement, a sensitivity toward which our perceptual apparatus is virtually programmed from the day we open our eyes. The brain is accustomed to dealing with movement which occurs through time on a moment-to-moment basis, but Marey's photos condense a series of these moments into one "time frame," allowing us a different way of looking at patterns in motion.

Canadian filmmaker Norman McLaren made *Pas De Deux* in 1967, a short in which ballet dancers were, through optical printing and other animation techniques, transformed into lovely and graceful tracings of human gesture. Again, direct photographic techniques were applied to "real" subjects but this time, by way of new technologies, the inherent beauty of movement was expressed in motion pictures. McLaren removed all detail from the images of the dancers, converted their figures to outlines, limited visual clues and allowed our eyes and brains to consider motion, pure and simple, as amplified by multiple printing techniques that were uniquely cinematic.

Many interesting experiments in motion perception were performed at Bell Laboratories in the 1960s and '70s. One of the most surprising discoveries came from an experiment wherein lights were mounted on the major joints (knees, elbows and the like) of a seated man. When the film was played back, the viewer initially saw only a set of random dots, but as the subject began to stand up his figure became recognizable within two or three film frames (a mere fraction of a second). This experiment demonstrated how well-tuned our vision is to the perception of human movement.

Recollections takes the idea of motion in art a step beyond that of McLaren's work and compels the viewer to become a participant in the piece. In one sense, this is a traditional view — any work of art should engage the viewer to interpret it, and we should each see something unique, based upon individual experience. This is certainly a mode of participation or even interaction — as one moves around, gets closer to or walks away from the artwork — but *Recollections* demands a bit more. If one remains immobile within its boundaries, only a plain silhouette of his or her outline will appear and, in some modes, one's image may disappear altogether. The viewer must complete the piece by moving, with these movements directly affecting multiple aspects of the image.

Each *Recollections* installation (and there have been nearly a dozen to date) contains unique software that causes different responses from participants. Typically, one installation will run a program with eight to ten different modes, each one lasting about 20 seconds. The speed, color sets, dissolve patterns, color dissolve rates and silhouette types are some of the elements that are pre-programmed. The spectrum for the current version has been chosen from a palette of 4096 colors.

The present hardware configuration of *Recollections* consists of a small single board 6502-based computer, a proprietary image-processor which I designed, an industrial-type video projector, a B/W video camera, and a special retro-reflective screen to produce a silhouette. The software has been written for the system using the programming language FORTH. The image on the screen is composed of a matrix of 256x242 dots known as pixels or picture elements or pels.

The image is received by the camera and goes to a one-bit A/D converter, which extracts the silhouette by differentiating the signal. In the current performance system, a 4-bit A/D converter is used. These data are then mapped into the video frame buffer and then remapped on output through a color look-up table. All operations done on a pixel-by-pixel basis are executed in hardware; all changes on a frame-by-frame basis are done in software.

I think *Recollections* is successful in creating an exciting interactive environment. Time and color elements linked to the participant's motion cause a noticeable behavioral change while he or she is experiencing time-altered space. One's perception of movement is, perhaps, subtly modified after encountering *Recollections*.

Acknowledgement

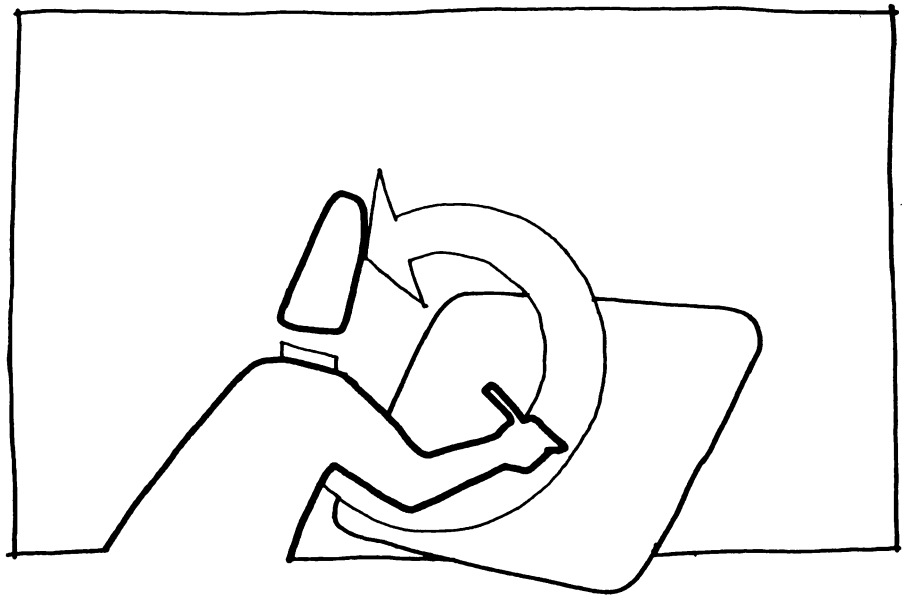
The author thanks the Fort Wayne Museum of Art for permission to reprint this material. It appeared in the booklet *The Other Television: Video by Artists*, September 27 - November 2, 1986.

Pictorial Conversation:
Design Considerations for Interactive Graphical Media

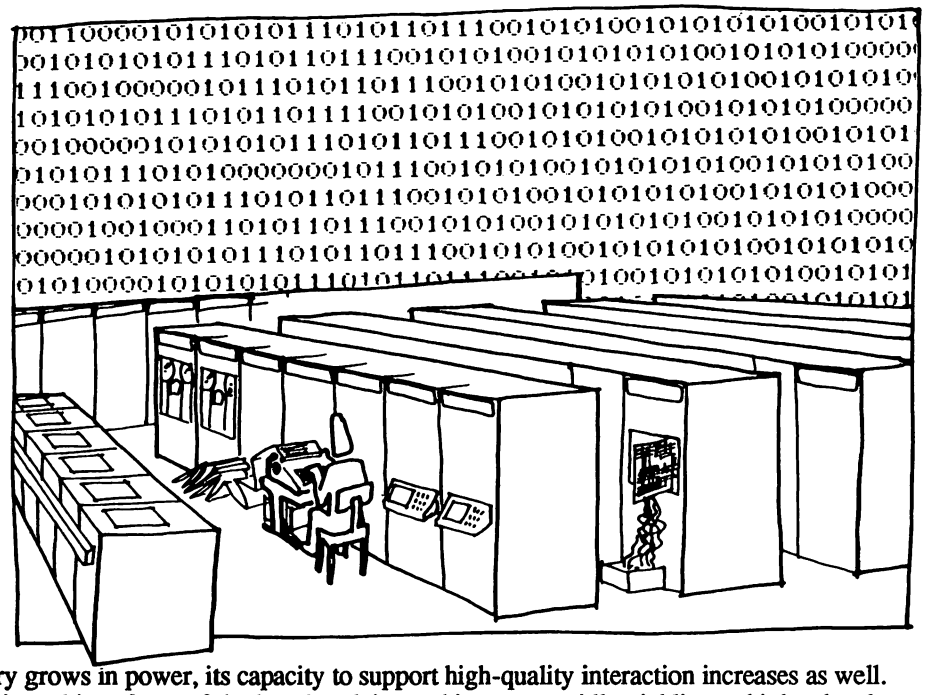
Rob Myers
Silicon Graphics Computer Systems
2011 Stierlin Road
Mountain View, CA 94043

Abstract

Considerations are presented for the use of interactive graphical representations as a medium for expressing, refining and conveying ideas. A conversational paradigm is introduced for image-based dialogues, and conventions are applied from other graphical contexts. Finally, a scenario is developed to illustrate these design considerations in application.

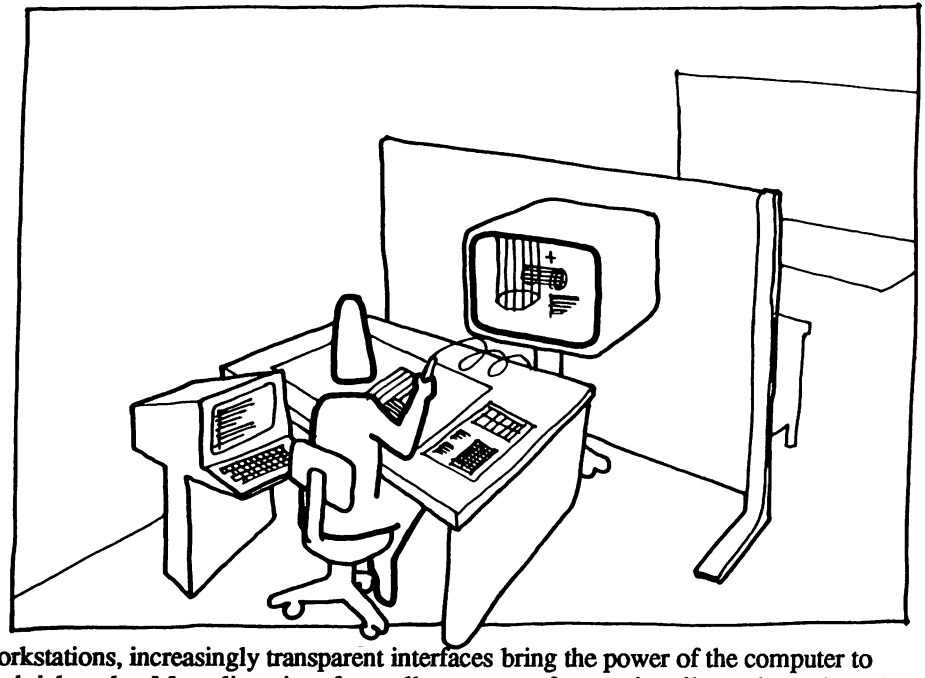


Beyond teletypewriters



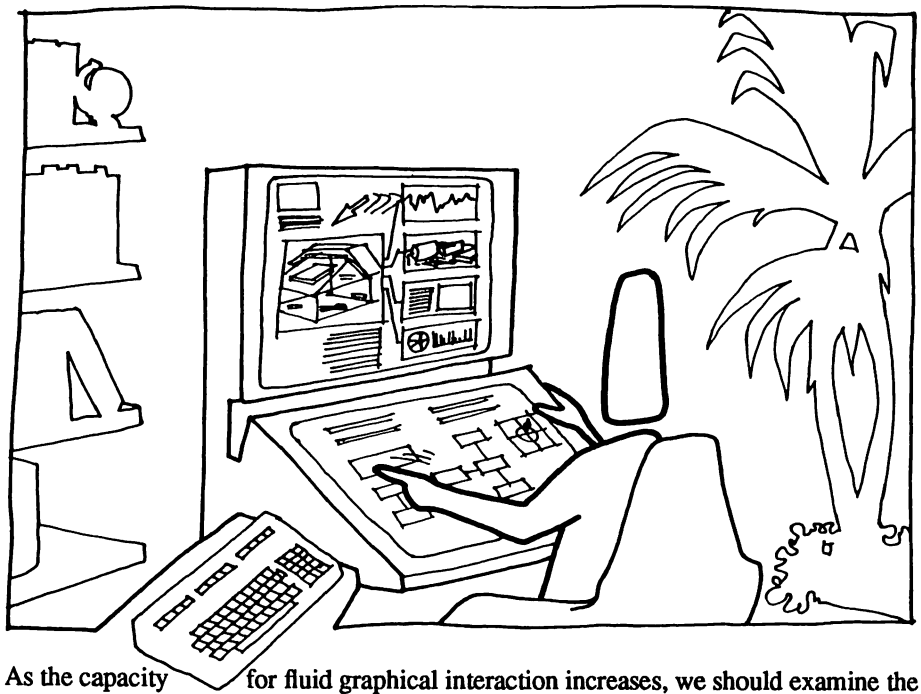
As computing machinery grows in power, its capacity to support high-quality interaction increases as well. The hostile, machine-oriented interfaces of the last decade's machines are rapidly yielding to higher-level exchanges, which utilize their users' efforts much more effectively. No longer is alphanumeric text the sole vehicle for transaction; a growing repertoire of visual and gestural techniques is extending the quality and character of human-computer communications.

Graphical interaction



On today's graphical workstations, increasingly transparent interfaces bring the power of the computer to bright people with a tough job to do. More direct interfaces allow users to focus primarily on the real work at hand, diverting less of their (costly) attention to the endless arcane trivia of the machine's inner workings.

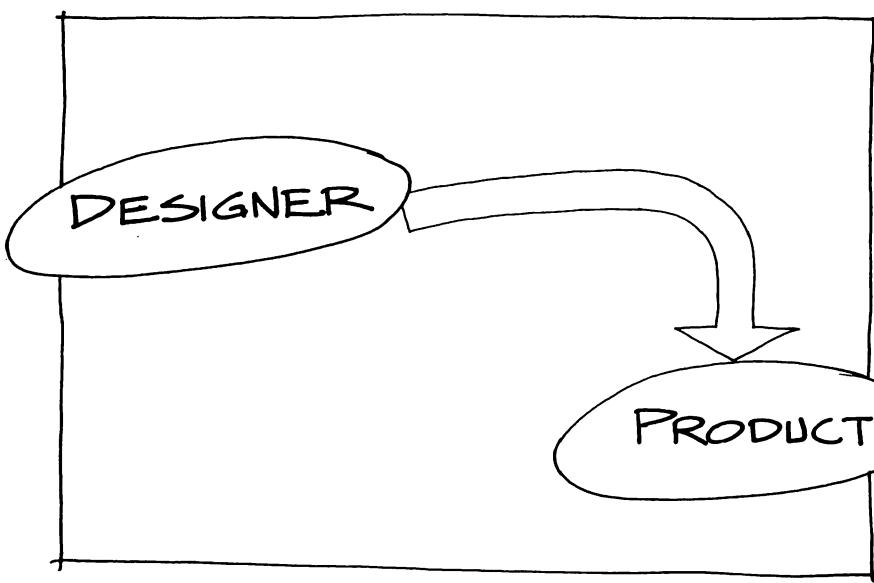
The payoff for preserving users' focus upon their actual tasks is becoming obvious in a widening arena of serious applications. This trend insures that an increasing percentage of the computer's expanding resources will be devoted to the implementation of much friendlier, much smarter, higher-bandwidth human interfaces.



Pictorial conversation

As the capacity for fluid graphical interaction increases, we should examine the nature and roles of the dialogue with visual representations. It is this process of pictorial conversation that provides the basis for the emerging interactive graphical media.

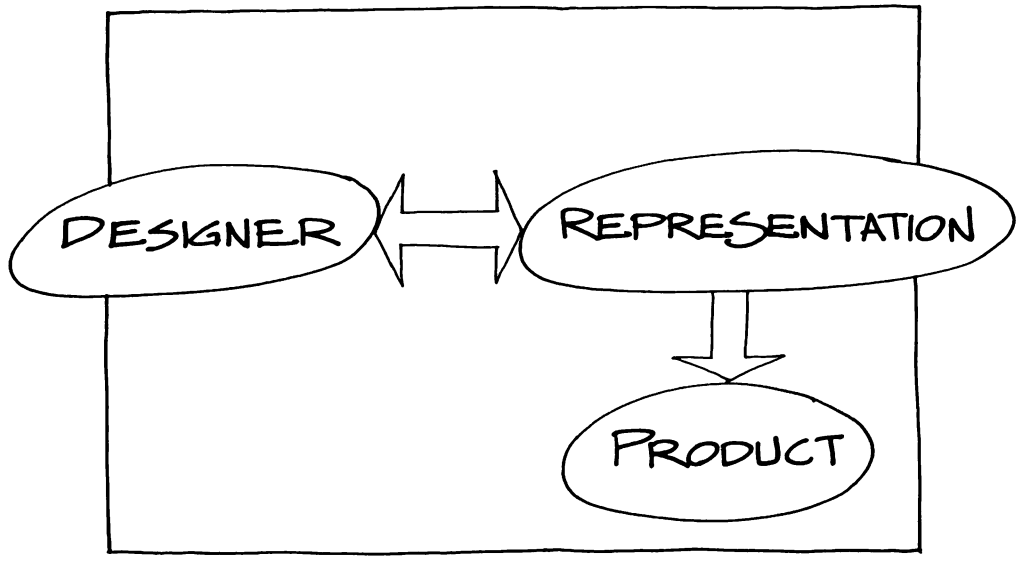
The following exploration of pictorial conversation tends to orient itself around the design process. This is primarily because there is a strong tradition in the use of visual representations in this field; concrete examples are readily obtained. Many other fields have their own visual foundations, and, under the new dynamic graphical media, many more visual ballparks will be established and extended.



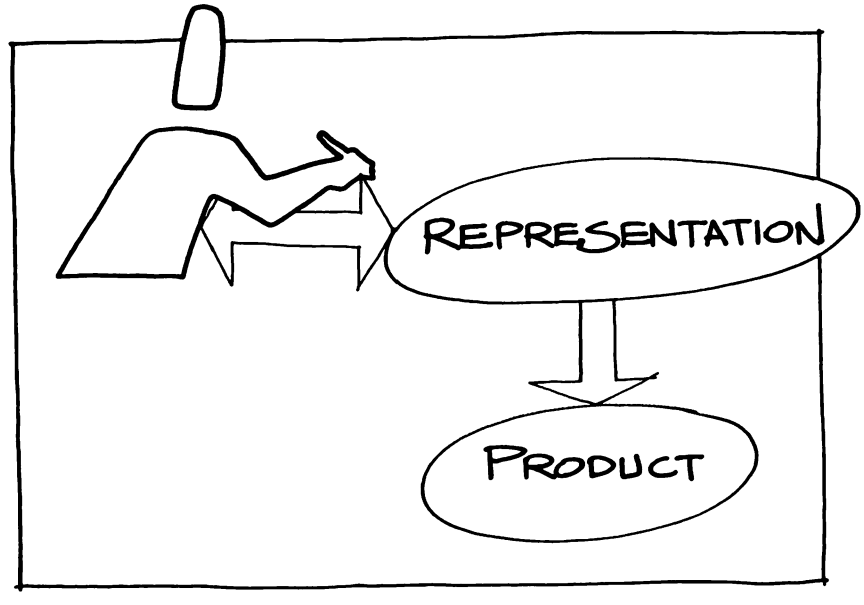
The design process

To get a look at the structure of pictorial conversation, then, let us first examine the design process. Here, the basic task is for the designer to define a finished product.

The design process



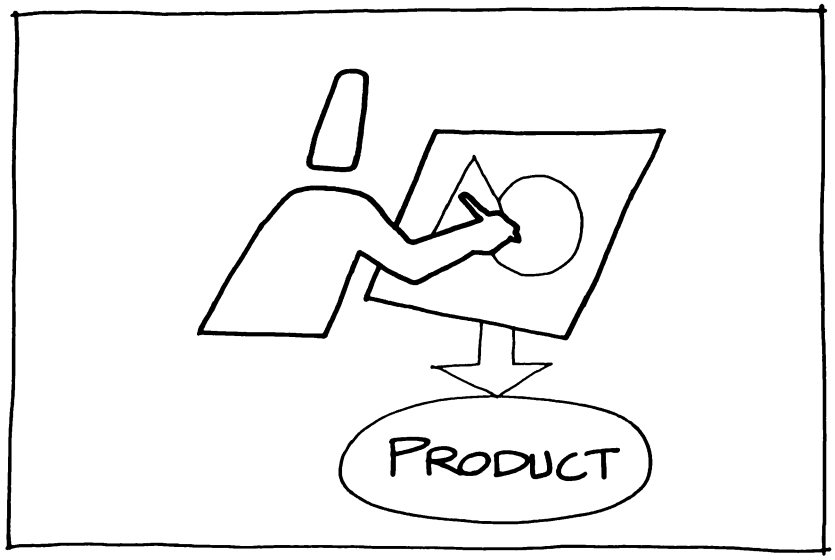
He does so by interacting with an interim representation of the product. This intermediate representation provides a more responsive, malleable medium for exploring and defining the sought-after end result.



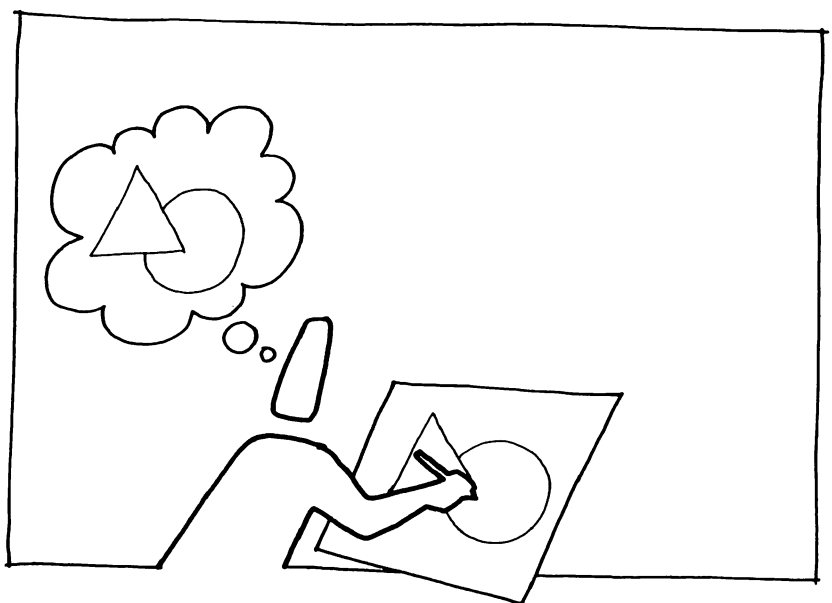
The use of an external representation helps the designer to formalize and manage his thoughts about the product:

to make the idea explicit,
to free up mental capability, conserving it for unresolved issues,
to encounter each design's concrete difficulties and advantages,
to employ spatial and rendering techniques for organization, and
to communicate the design to others.

The design process

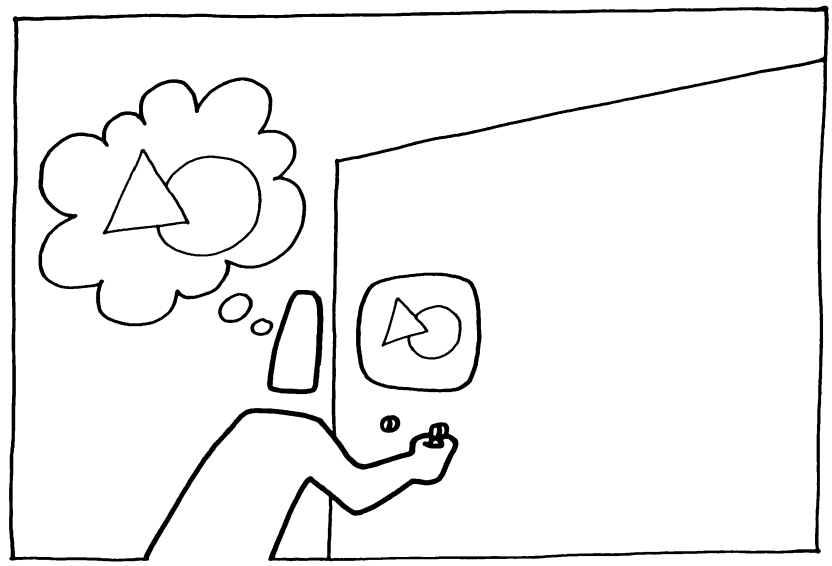


Here, then, we have the designer working directly with a graphical representation, articulating and evaluating many possibilities on the way to what the product will finally be.



Looking closer, what is going on is a feedback process, where the designer tries to capture what he is thinking about in a concrete form, the external representation. In turn, his perception of the current representation affects the form of his dynamic internal representation. And so the process iterates towards stable solutions.

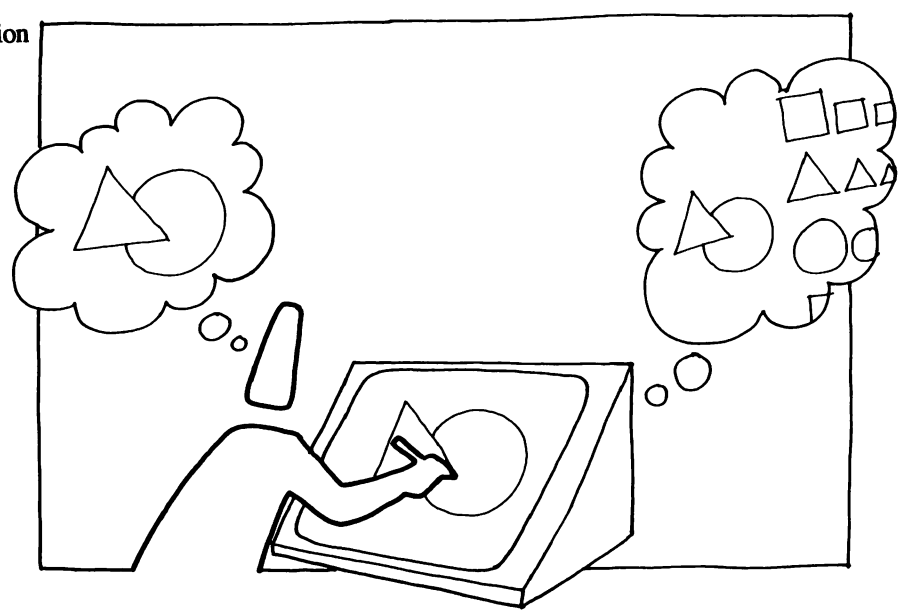
Computer graphics as a barrier



Traditionally, computer graphics has given the designer an increasingly malleable representation to work with. However, conventional systems cannot match the expressive articulation of "old fashioned" paper media, cramming the designer through a narrow porthole of limited interaction.

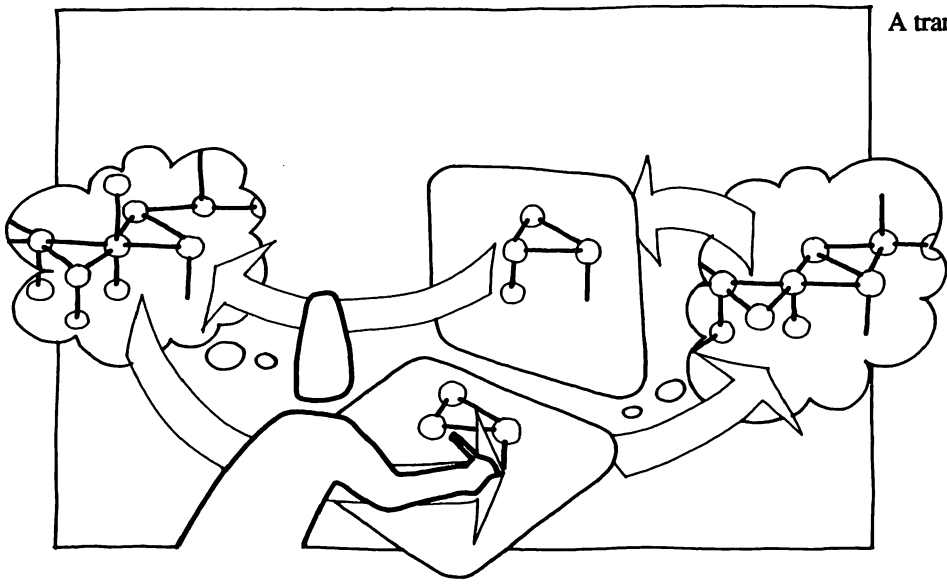
All too often, the real creative work has already been sketched out in front of the screen, on a rumpled sheet of flimsy paper. This rough is laboriously transcribed into the machine, to take advantage of the computer's ability to produce precise, easily-changed drawings. The computer is not being used to support the ideation process.

Computer graphics as an extension



As new interactive graphics systems grow up to the task, however, we can focus upon graphical representations not just as finished, "pretty" products themselves, but as a medium for expressing, molding, capturing, and conveying ideas.

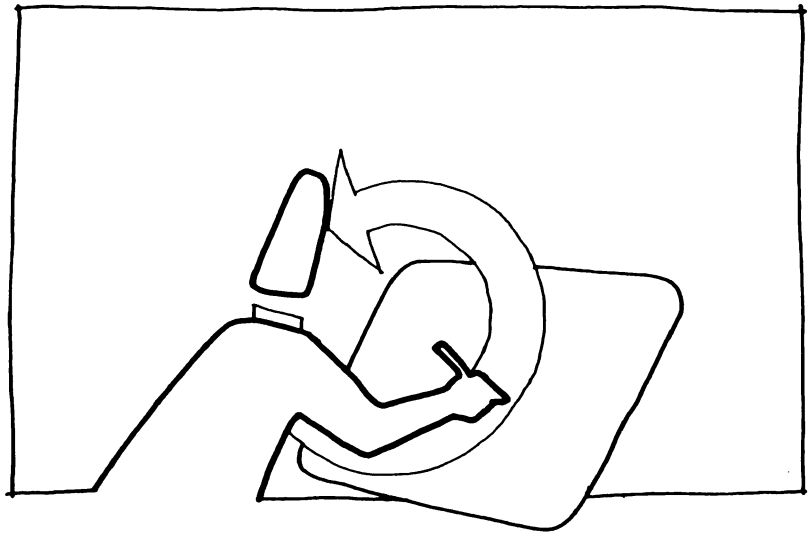
Assuming a system as agile as pencil and paper, let's look at the power computer-based graphics bring to bear. No longer is the dialogue just between the designer's mental representation and the concrete representation in front of him. The computer's internal representation adds another important dimension to the design process.



A translation process

Note that there will not be a one-to-one correspondence between the various representations throughout this process. The designer's internal rep will never be identical to the form he is able to express to the machine. In turn, this expressed representation undergoes another transformation when captured as the machine's internal rep. And, of course, this is not the same representation that the machine feeds back to the designer. This presentation by the machine is then further interpreted by the designer's perceptions.

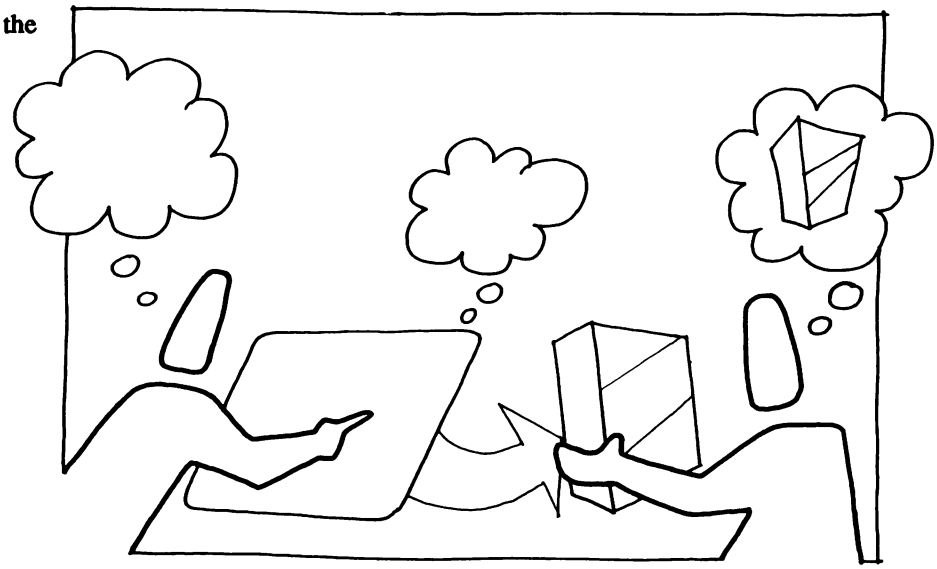
These mappings are of central consideration in defining a medium which supports pictorial conversation.



A direct experience

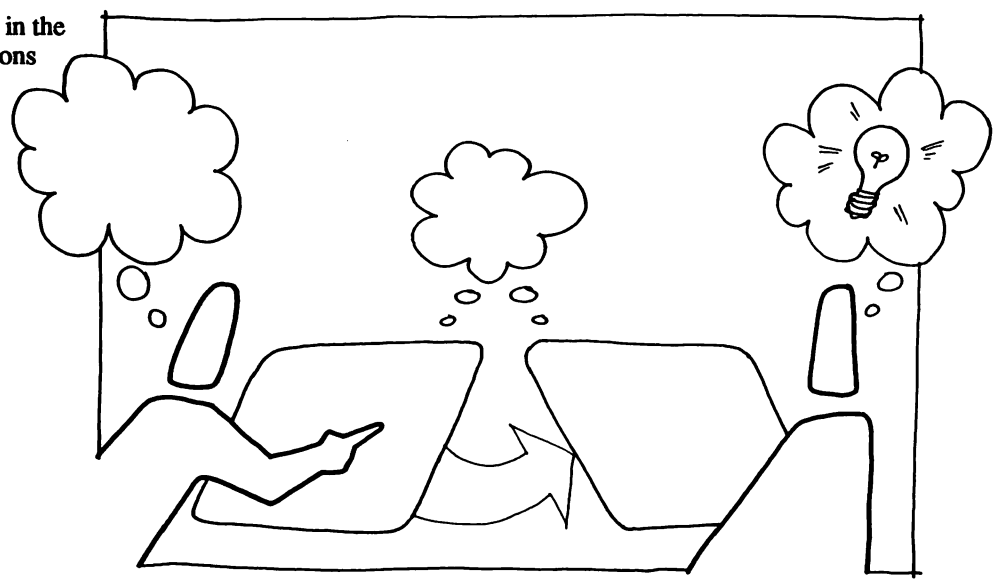
However, from the user/designer's point of view, the entire process must be perceived to be as fluid and as direct as a good sheet of sketch paper. The mechanics of the medium must not intrude upon the momentum of the conversation process.

Pictorial conversations in the product design domain

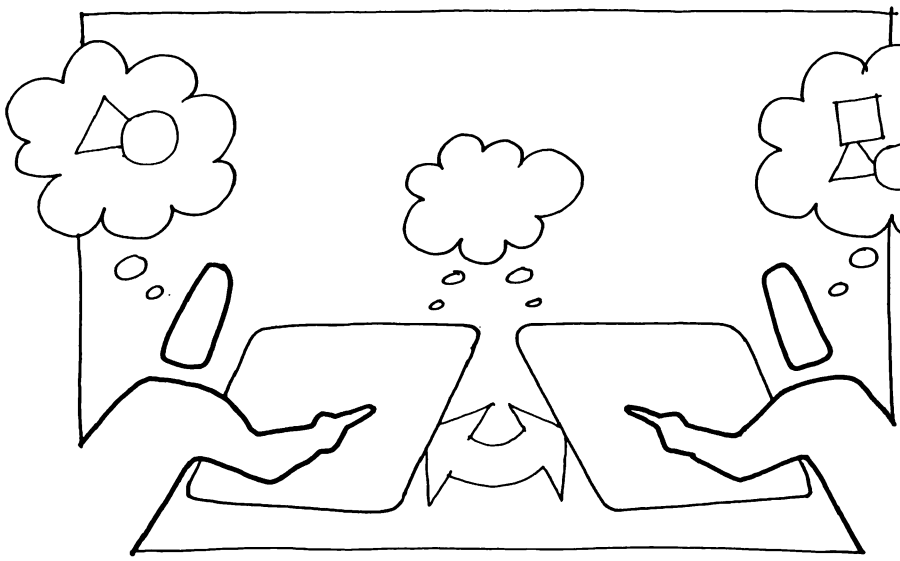


The finished product design will be a success if it is accepted by consumers. In the product design field, then, one might say that the outcome of the pictorial conversation process is to produce a specific perception in the consumer's mind.

Pictorial conversations in the design of communications



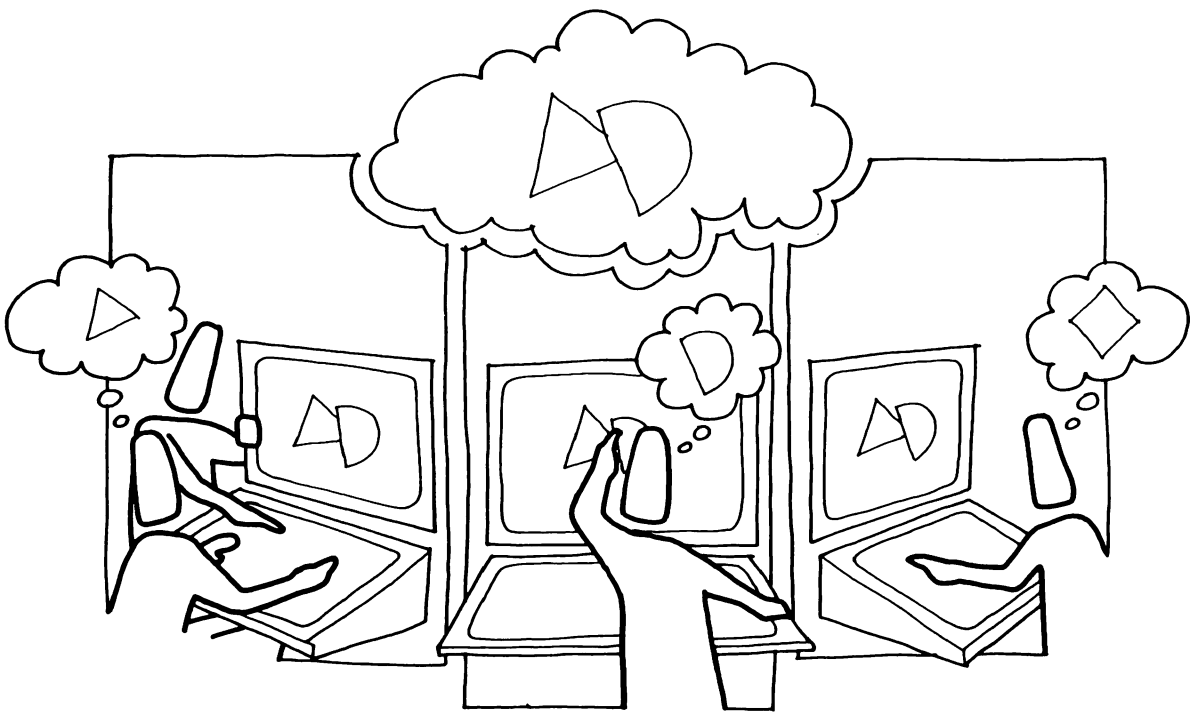
In the field of education, the goal is also to change mental states in the final consumer (student). However, the educational process employs the interactive, pictorial representations directly, as the vehicles of perception themselves. The end-users of the communications product have direct contact with the results of the pictorial conversation development process.



Pictorial conversations in the development of ideas

When the pictorial conversation systems are linked with a full, two-way flow of information, participants can influence each other directly, evolving mental states together, influencing and developing ideas via the wide bandwidth of the visual interface.

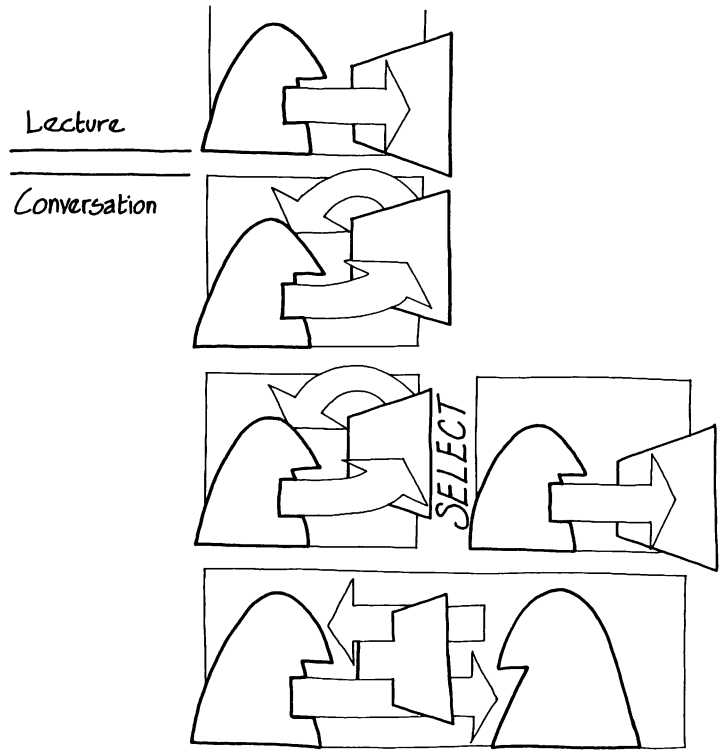
It seems to be a natural application to use such systems for designing systems to design systems to . . .



In a community of such linked interfaces, the system can be used to explore, build, and modify cumulative representations in addition to the more familiar personal ones.

The nature of the dialogue on such systems can be explored by examining the forms of familiar, face-to-face human conversation.

Forms of conversation

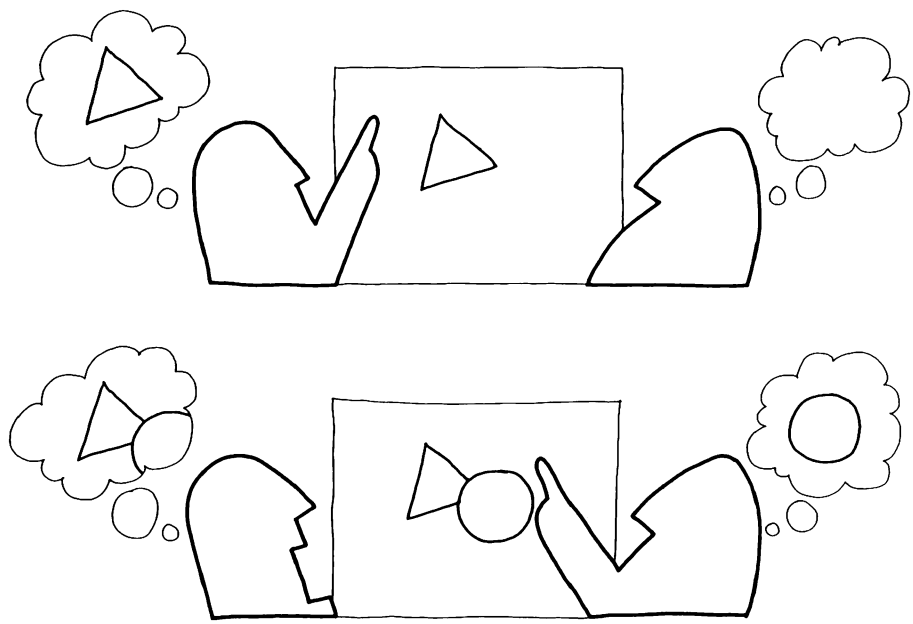


Lecture is predominated by an outbound flow of information. This form does not include much of the input and interchange which characterizes conversation.

Self-articulation entails the externalization of new or revised points, and feedback in response to the building cumulative representation. "Thinking out loud/on paper" concretizes and refines ideas.

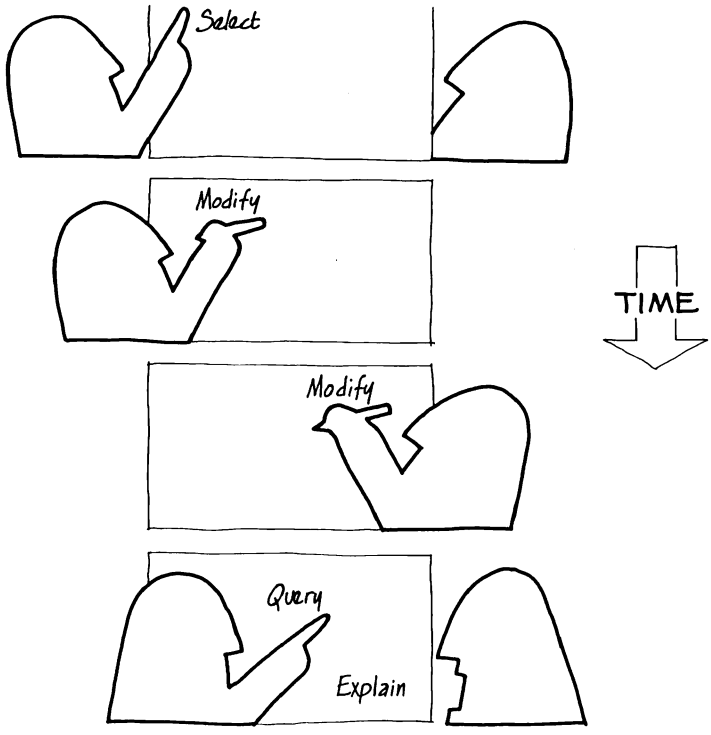
Presentation employs self-interactive development and selection of the content, followed by a more formal delivery of matured concepts.

Dialogue involves a two-way flow of information, with interruptions, inquiries, formal and informal feedback, and a shared, changing intermediate representation.



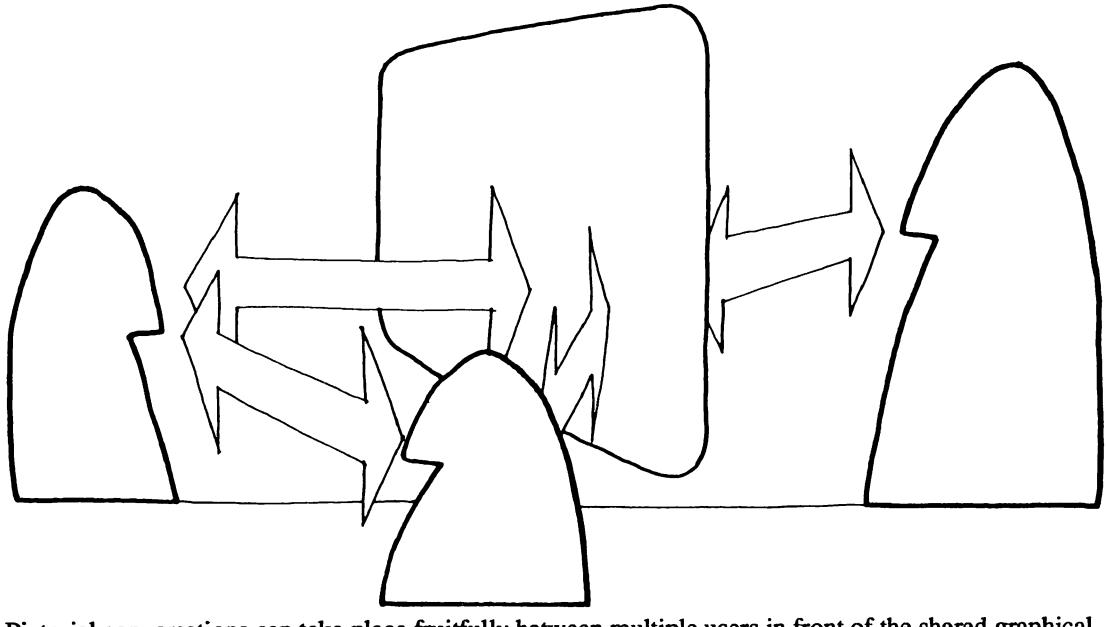
Pictorial dialogue involves the composition of a cumulative representation over time, with the participation of multiple parties. It entails not only the articulation of ideas out into the common pool, but also the integration of new points back into the participants' internal representations.

The pace and the give and take of information in the exchange will vary over time and by participant, as will the methods used to acquire and yield the "floor".



Over the course of the conversation, the graphical representation will serve in a number of roles:

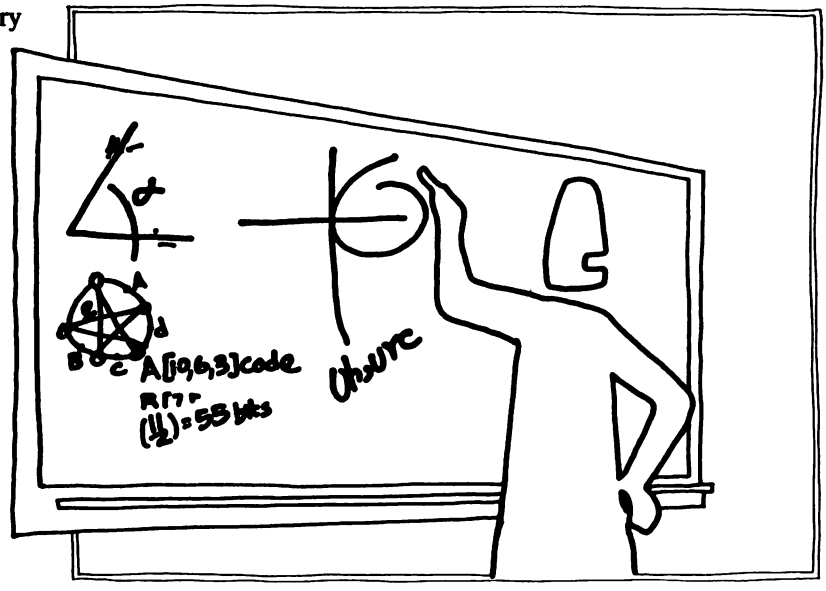
to select and identify domains of discussion during the conversation,
to provide reminders or visual placeholders for further discussion,
to serve as a foundation for modifications by all parties,
to maintain alternate forms of the same or divergent ideas, and
to serve as a concrete referent for queries and explanations.



Pictorial conversations can take place fruitfully between multiple users in front of the shared graphical representation, as well as through the representation to remote participants or off-line authors.

Metaphors for the use of imagery in conversation

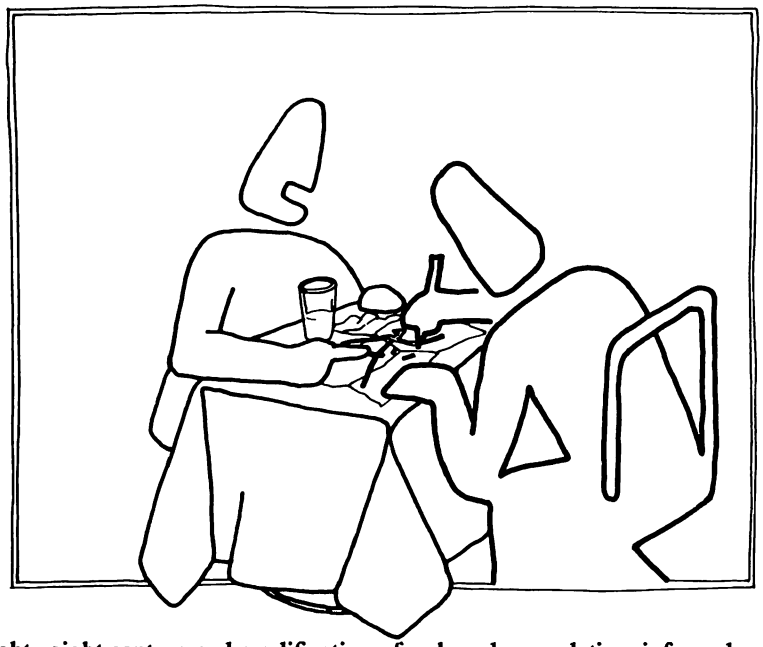
Chalk talks



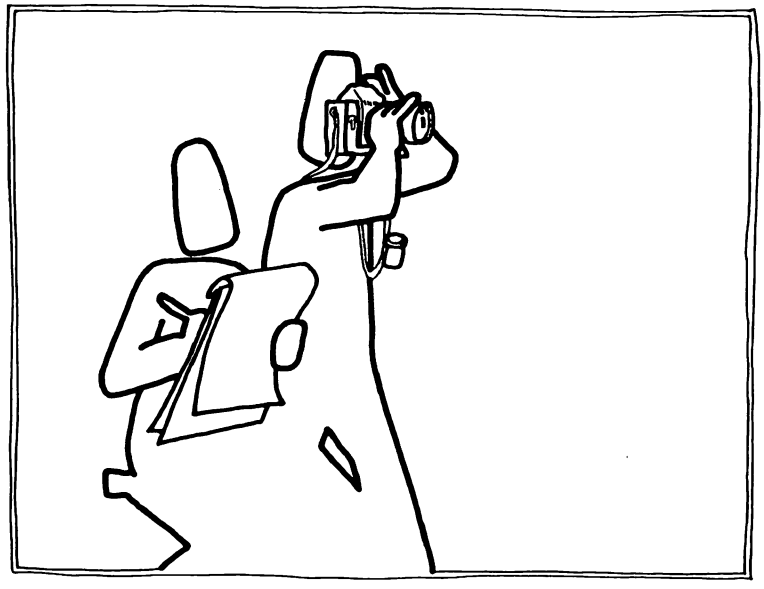
A number of metaphors can be drawn from traditional media, to discuss the use of imagery in a pictorial conversation environment. Those presented here offer alternatives beyond the well-established drafting table and messy-desktop metaphors.

Chalk talk representations provide evolving referents during a discussion. These transient forms will be of little use to someone who was not present during their evolution, but they have served the participants well, providing landmarks, visual referents, and placeholders over the course of the discussion.

The design lunch

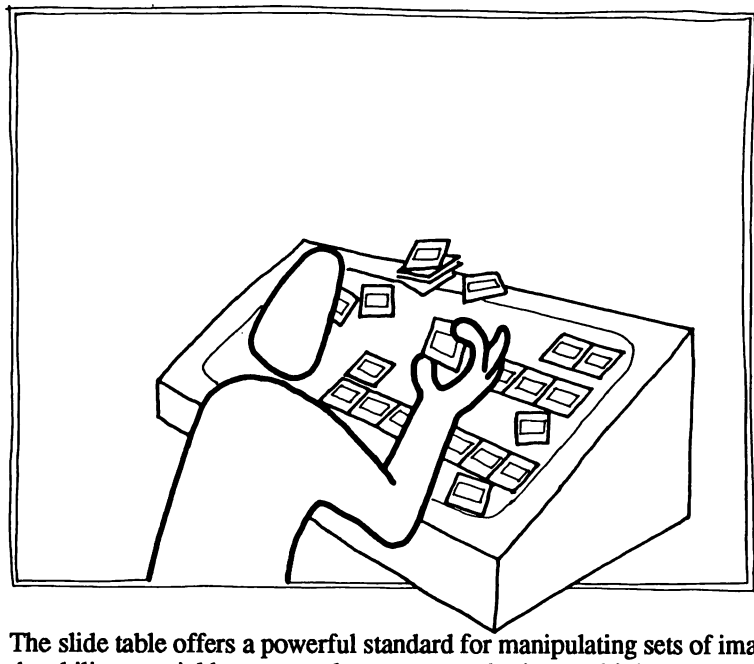


Napkin representations support the lightweight capture and modification of a shared, cumulative, informal image during a sudden brainstorm. Such images should almost certainly be promoted within a few days, before their meaning fades from even those who were there firsthand.



Images of the real world

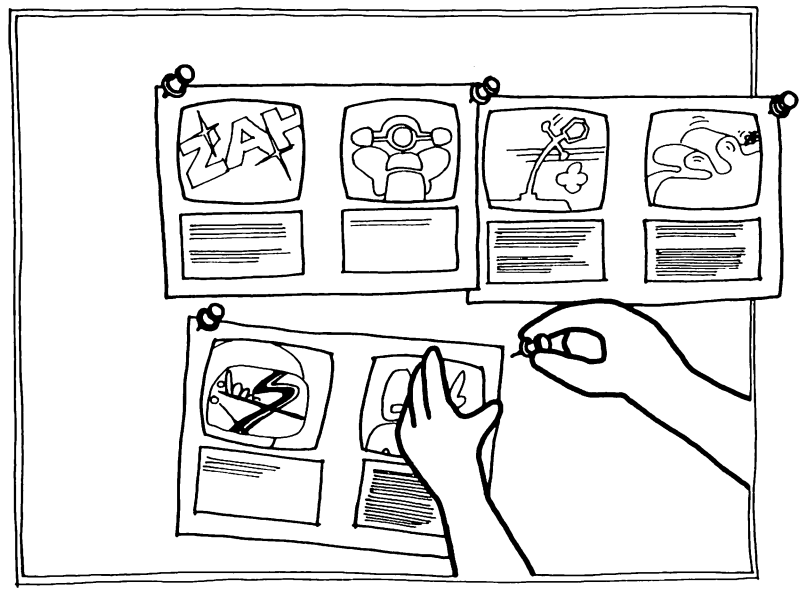
Likenesses are important residents in a pictorial conversation system, providing a snapshot or rendering of something in the real world. Photographed or rendered images can capture an inspiration, a reminder, or quick study in ways that diagrammatic or abstract representations cannot. It is important to casually introduce images from the world into the machine, just as it is important to casually output images from the machine into the real world.



The slide table

The slide table offers a powerful standard for manipulating sets of images. Pictorial conversation requires the ability to quickly sort, catalog, group, and reject multiple representations. Note that traditional slide table usage provides good overviewing combined with a nimble "zoom in" when the user desires more detail.

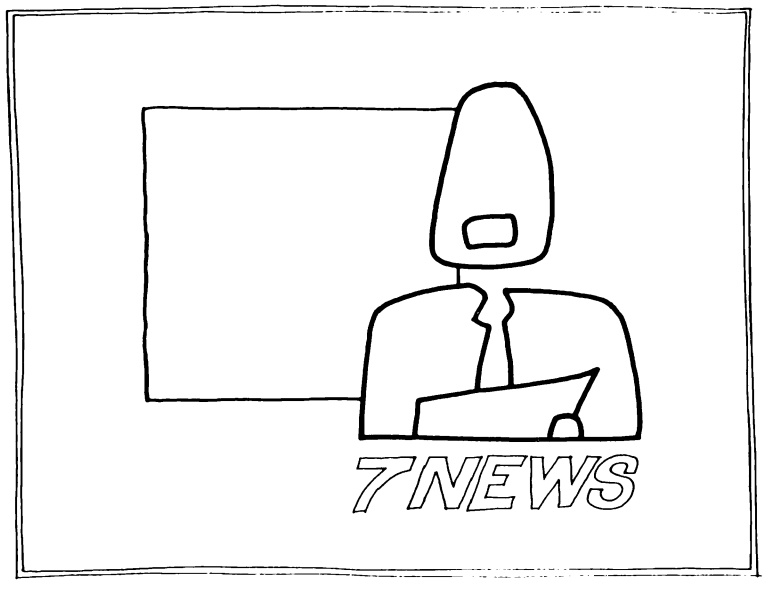
Storyboards



A storyboard collapses motion sequences into a series of key, summary images. This provides a useful model for keeping track of dynamic pictures, animations and movies on a crowded worksurface at a (static) glance. In addition, the storyboard's conventional role in planning and previewing sequences before they have been produced is appropriate in this setting. The accompanying annotation and directions offer another dimension to cataloging and managing moving picture sequences.

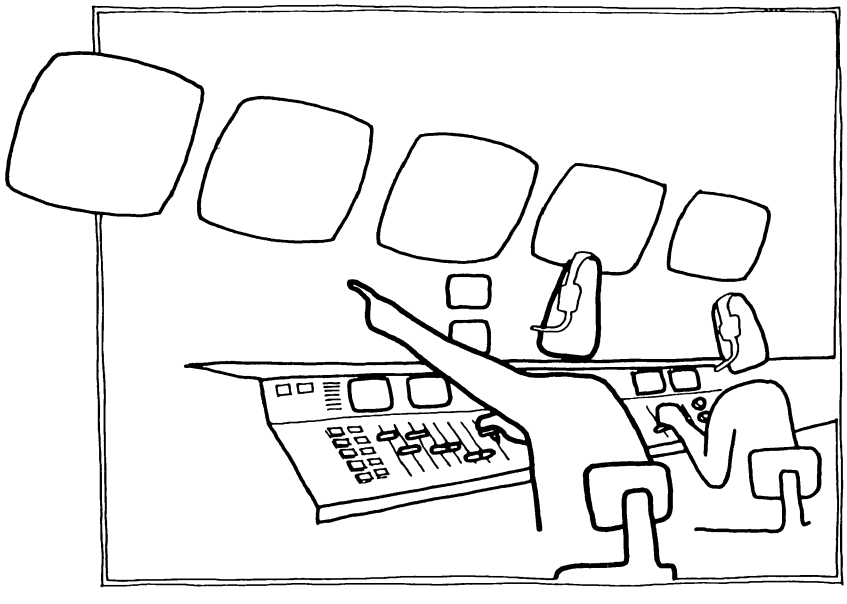
The editing room

Videotape editing offers metaphors for controlling the computer's ability to re-sequence and composite multiple motion clips into new presentations. Timing and visual continuity are just two of the editing-room skills relevant to the pictorial conversationalist faced with presenting sequences of dynamic images coherently.



Newscast continuity

On the flip side of managing motion segments, alternatives to formal editing disciplines do exist. An excellent example of this is "anchorman glue"; the use of voice-over narration to provide continuity across completely disjoint film clips, as brought to you by the evening news.



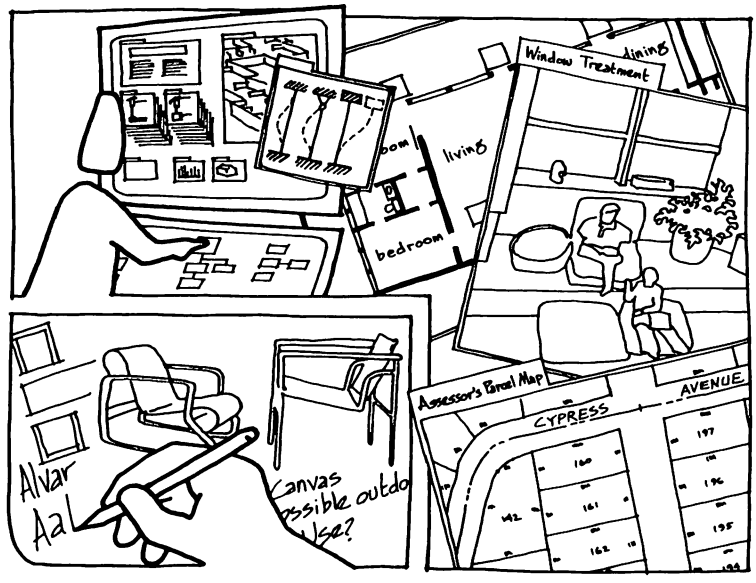
Live sports coverage

In an environment connected to numerous sources of active information, a relevant model may be found in the Technical Director orchestrating a televised sports event. Here, the art consists of channeling multiple "live" sources into a single, coherent, focussed stream of information.

Finally, to examine the principles of pictorial conversation in application, let's revisit the design process. The following case study illustrates a range of methods for employing visual representations, using an architectural design scenario.

An architectural design scenario

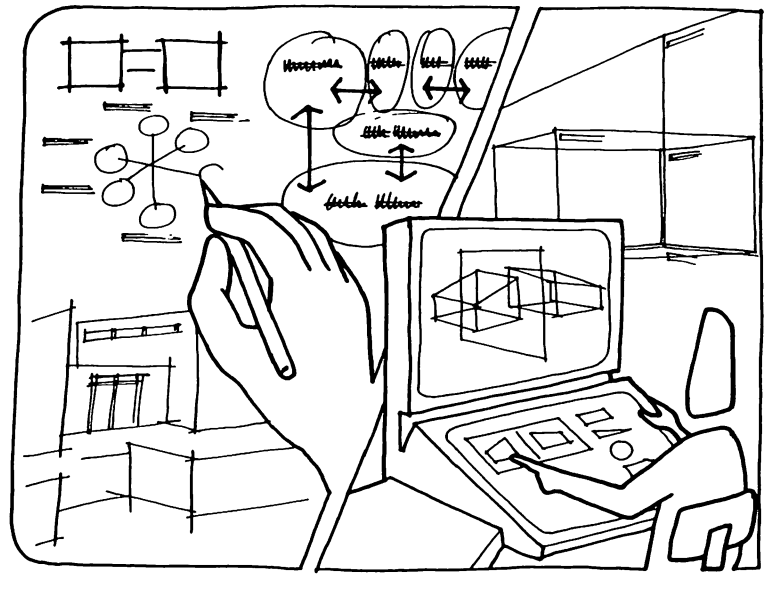
Building a background for ideation



Conducting preliminary research, the designer gathers and browses imagery and information sources. The collection process involves a relational search of various image caches, sorting useful and interesting pictures and data, and interrelating them into accessible groupings. Information collected includes applicable structural and mechanical properties, specific site data, relevant past work, and inspirational or idea-trapping photos.

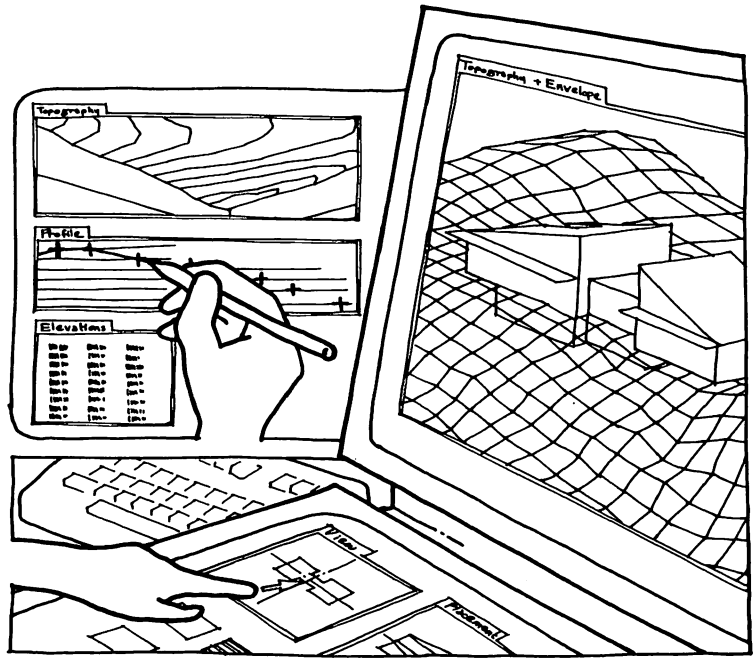
As the designer organizes the material for study, the found images can be annotated and modified, just for their instance in this project.

Capturing the expression



Working from the background towards a concept, the designer employs loose, freehand, highly changeable representations, like a sketchbook with enhanced power to juxtapose, combine, dismiss, or firm up images.

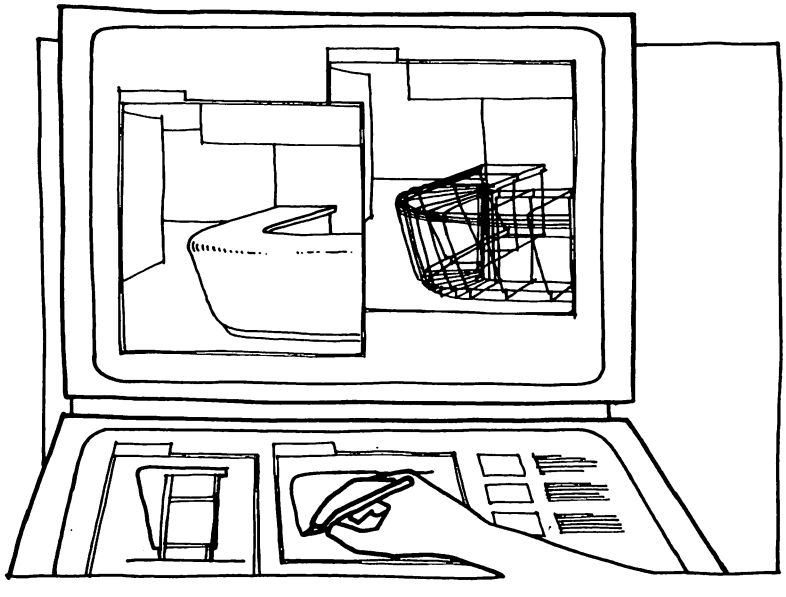
Progressing into initial decisions, the designer proposes tentative models in a sculptural fashion. This transition entails working closely back and forth with the sketchbook, blocking out forms and giving flesh to concepts, comparing 2d and 3d aspects of the emerging form against the design criteria continuously.



Employing multiple representations

Preparing to match his developing building model to the site, the designer generates a graphical instance of existing topographic data. The resultant contour model can be adjusted by drawing modified profiles, which continuously update the map being viewed as well as recording the new grading information.

The combined representation, incorporating the building model with the terrain model, is viewed, examined, and tuned in response to various proposals. This interaction framework uses tightly-coupled, parallel, alternative methods of visualizing and interacting with a single underlying model.

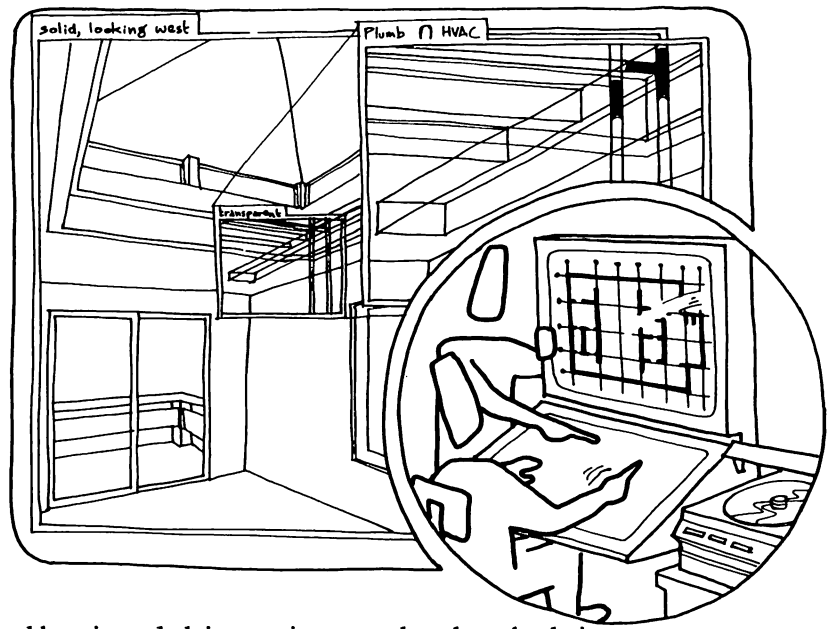


Interactive geometric modeling

Passing a cabinet section through a path on the floor plan, the designer generates a 3d model of the reception area. The complex form can be investigated and revised in continuous interaction, until all criteria are satisfied.

Successively refined renderings are employed to allow tradeoffs between the fluidity of interaction and quality of the presentation. A lightweight wire-frame view continuously tracks all modifications to the model, while the solid view fills itself in with an increasingly realistic rendering when the incoming changes slack off.

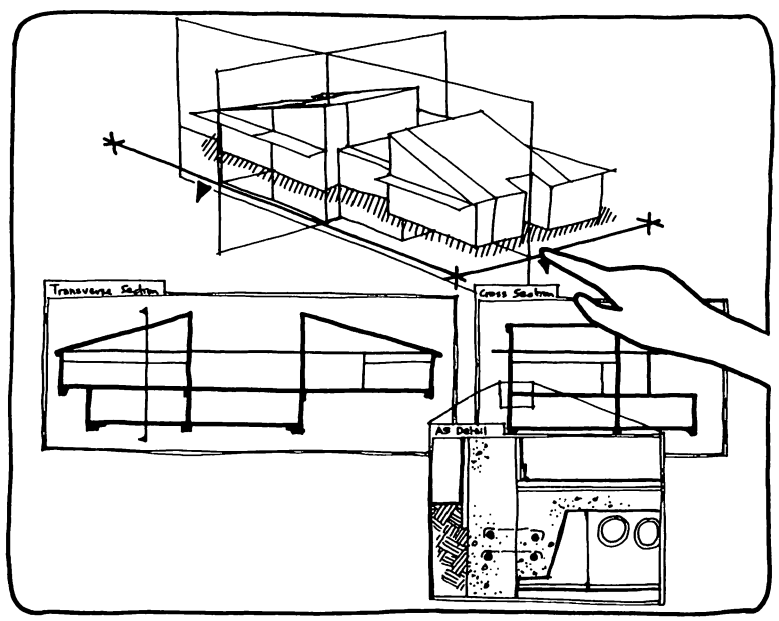
Image operators and virtual tools



In studying the building's geometry and how its underlying service networks relate, the designer can "stroll" about the model. Normally viewing a solid rendering of the model, a overlaid window with controls to set to various degrees of transparency can be employed to examine how things are working underneath the surface. Volumetric boolean arithmetic can be invoked to bring conflicts to light. Here, the plumbing network is intersected with the heating system, highlighting a spatial collision between them.

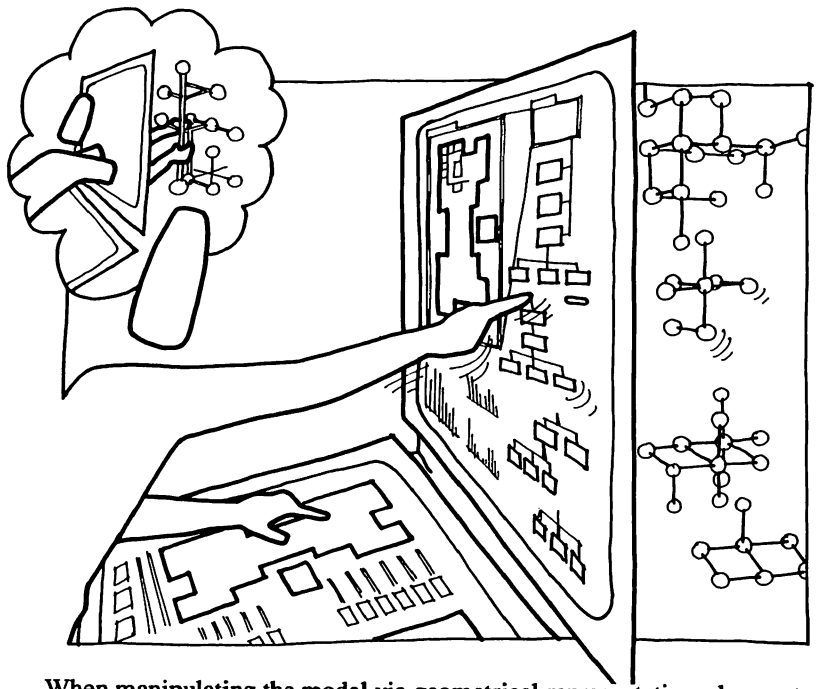
In this usage, windows operate as filters which present the underlying representation via their own methods (e.g renderers, X-ray viewers, magnifiers). Such versatile software tools can be considered to be analogous to the designer's familiar physical tools, where specific gestures and utility views take the place of electric erasers, architectural scales, and the like.

Spatial indexing of information



Using a 3d reference view, sections of the model can be taken at will. As the slicing plane is located, the representation of the building section at that point is simultaneously updated. This section is more than a "picture", it allows access to the internal model. Thus alterations to the section simultaneously alter the model itself, as well as all other connected views.

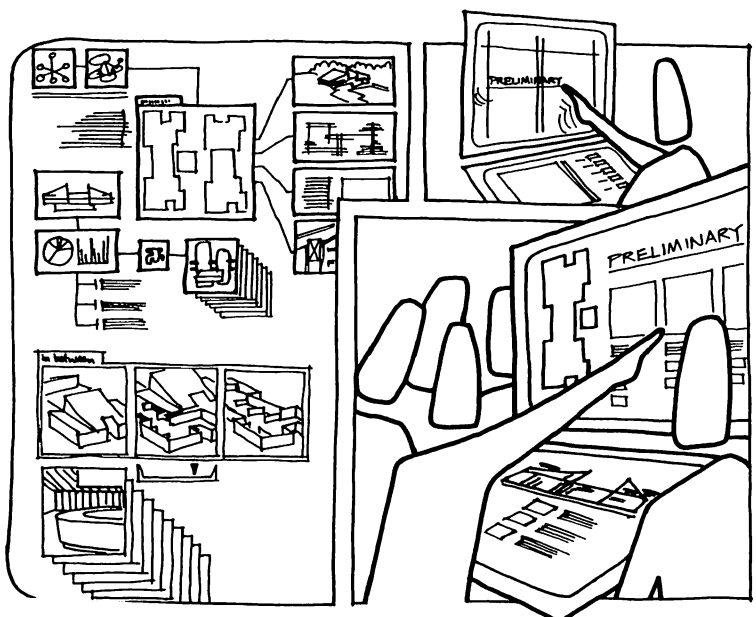
This hierarchy of "images that work" extends down to the level of detailing, where explicit treatments can be zoomed in upon, changed locally or globally, and then re-buried until they are again called up for attention. This approach invokes the use of concrete spatial metaphors to access information in a coherent, familiar fashion.



Literal presentation of internal structure

When manipulating the model via geometrical representations does not support the kind of overview power possible in the relationally defined data base, the designer can employ more abstract tools for pictorial manipulation. Here, a much more literal representation of the machine's internal structure is displayed, allowing the underlying rep to be accessed via "hands-on" manipulation, as opposed to indirect, metaphoric means.

By directly interfacing with the structure of the model itself, entire families of options can be connected, nested, re-attributed, or set aside for possible future use. The designer can bring the most powerful options and variables to the "surface" of the representation in the form of buttons or windows, accessible side by side with other established controls.



Composition tools for finished presentation

The designer can employ the entire bag of graphical tricks in preparing a display suitable for presentation. While his working displays may be idiosyncratic, personalized, or "quick and dirty", the front end representation may be spruced up and made more communicative and effective for an outside audience. Captured explorations can be ordered for coherent access. Animated sequences may be prepared to improve continuity when key transformations are demonstrated. Key views and workspaces are laid out across the full face of the display and combined with legending and landmarking cues.

Finally, the presentation package is performed, with both the designer and the client involved in making their ideas explicit, using the specifically developed image bank to communicate about the proposal.

Plasm: A Fish Sample

Rob Myers
Peter Broadwell
Robin Schaufler

Reactive Graphics Installation
SIGGRAPH 1985
SIGGRAPH 1986

Interactive Art

Goldfish Putty

Remember *silly putty*, that stuff someone gave you once a long time ago? It immediately took your fancy, you smushed it out, balled it up, bounced it around and generaly had fun with it.

Imagine that give and take relationship you had with *silly putty* and drop it into the computer. Here, the stuff can become responsive in many new ways.

Fishtory

The challenge is to develop an interactive environment which manifests a convincing semblance of behavior. This involves integration of real-time image manipulation, motion representation, behavior modeling and user interface design.

The aquarium framework for interactive environments was conceived and developed by Ann Marion at Atari Sunnyvale Research. A coral reef aquarium could take advantage of the exaggerated symbiosis commonly used to teach about ecosystems.

Meanwhile, back at the Mathematics Imagery Group, Rob Myers and Peter Broadwell were experimenting with interactive particle systems fireworks. This blossomed into interactive fishworks, inspired by Alan Kay's SIGGRAPH '84 call for behavior-based graphics, along the lines of the aquarium's fish putty. Plasm's behavior modeling system builds upon Robin Schaufler's system to support object oriented design in "C".

The Silicon Graphics IRIS workstation with custom graphics hardware and sufficient computing power provides an appropriate environment in which to develop interactive behavior systems. The evolution of machines and software technologies which can support this work is relatively recent and we applaud it.

Reactive Graphics

Animated images can be interactive, reacting to one another and responding to prodding from the outside world. Faced with such responsive imagery, viewers become participants, actively exploring the flows and balances in the display. Pictures like this are based upon a deeper underlying structure, one which describes not only the geometry, not only the motions, but also the perceptions and reactions of the players within the scene.

Not A Script

Unlike the direct, predetermined control of a script, the interaction designer exercises a more indirect control over her creation. While the animator realizes a single, precisely defined motion clip, the interaction designer produces a vocabulary of possible motions.

Four basic levels of interaction with this vocabulary can be outlined.

> The first, passive level finds no user input forthcoming. Here, the ecosystem is a closed loop of intelligent interactions among members of the environment. No new information is created or destroyed as the objects follow their behavior.
>
> At a slightly higher level the observer can reach in and affect parameters that influence an objects behavior. Furthermore, she can mix and match behaviors from the kit. This implies some information flow into the system.
>
> The third, rule-building level, allows the user to construct brand new behaviors and relationships on the fly, adding to the kit of available behaviors and interactions as desired.
>
> The addition of adaptive behaviors realizes a fourth level, where interactions among the players themselves actually modify the behaviors. Such adaptive behaviors would allow fish to be "trained" by their experiences. And what about objects which generate their own behaviors?

A fully realized system implementing aspects of each level would be a very nice piece of putty to play with. If the plasm really takes on making up its own behaviors, perhaps it would start playing with the observer.

Towards a Useful Description Language

How do you describe behavior? Initially we have described it using conventional programming languages. As an animator thinks in visual terms, however, a visual and gestural language would more closely model her thought process.

A visual language might consist of rehearsal; rehearsing perception of a stimulus, rehearsing a response, and establishing causality between them.

To create such a language, we must have a meta-language to talk about it, much as we use grammars to describe conventional programming language. Want to help?

Art

Interaction of living things has always been a subject for Art, but outside of theatre, the computer is the first permanent medium capable of supporting an interactive first person experience. The viewer can be actively engaged in a feedback loop with the art object. Insights into complex dynamic phenomenon may be more easily communicated. The emphasis of design or artistic control is in creating underlying rules which will govern the behavior of the interactive experience. This will extend artistic powers of expression.

Determinism

A rule based system is deterministic, yet can be rich in complex behavior which need not be completely predictable. This apparently paradoxical notion is very much related to the realizations the computer graphics community is coming to with consideration of fractals and stochastic techniques for data amplification. It is possible to characterize complexity with far greater sophistication than was thought possible in the days when we thought there was nothing between regularity and randomness.

Peter Broadwell	...!ucbvax!sgi!peter
Rob Myers	...!ucbvax!sgi!rob
Robin Schaufler	...!ucbvax!sun!robin

A Multi-Representation, Bitmap Interface to the UNIX® File System Constructed from Cooperating Processes

C. D. Blewett, J. T. Edmark, J. I. Helfman, and M. Wish
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

FM is a prototype file manager which can be used for such purposes as browsing, reorganizing, cleaning up, and modifying UNIX directories and files. It is comprised of two new programs, *graphmod* and *dirmod*, along with Pike's *sam* text editor. The associated processes work together by means of another program which provides a novel type of interprocess communication (ipc).

The ipc module works in harmony with Pike's graphics process model, with asynchronous message passing. Messages are delivered in *fifo* order, but may be selected by the receiving process by tag, generic id, and specific process. Adding message handling to an existing program can be accomplished by adding a few lines of code and a switch to sort through the messages of interest.

The overall design of FM adheres to the UNIX tradition of building small, independent programs that can be simply integrated into larger systems. *Graphmod* provides a graphical interface for providing an instantaneous picture of all directories the user finds relevant at a particular time or for a particular task; *dirmod* allows the user to edit file attributes, such as permissions and owner, and to execute UNIX commands on specified files. Although these two programs, like *sam*, can be used completely independently, the beauty of *FM* lies largely in how they cooperate in creating a coherent representation of the working environment.

Introduction

The present research is focussed on the development of interactive bitmap graphics tools that enhance the interface and computing environment for general UNIX users as well as application developers. It adheres to the UNIX tradition of building small, independent programs that can be simply integrated into larger systems.

The specific prototype we have implemented [1] is called *FM*, for *F*ile *M*anager, since an important use is for displaying, reorganizing, cleaning up, and modifying UNIX directories and files. At the present time, *FM* runs under UNIX Version 9 on a 5620 Dot Mapped Display terminal in the *mux* environment [5,6], but we intend to port it to other systems. Although currently at a primitive stage, an eventual aim is to develop a visual interface for a range of UNIX applications that incorporates direct manipulation and control by the user, as well as improved feedback and visualizability.

General Design

FM is comprised of three programs, *graphmod*, *dirmod*, and *sam* [7], respectively, for displaying and manipulating (1) a graph (or subgraph) of directories in the file system, (2) a set of electronic forms associated with individual directories, and (3) the text in several files. The three processes work together by means of another program which provides a novel type of

1. Not all of the bitmap capability is implemented yet.

interprocess communication. The overall structure provided by these cooperating processes is intended to represent a three-level working environment which the user finds relevant for performing particular tasks (e.g., browsing, reorganizing, directory and file editing).

It should be noted that modification, or even substitution, of one of these modules (e.g., use of *vi* or *emacs* rather than *sam*) has little or no effect on the others. This greatly simplifies development and maintenance of the system, going well beyond the usual conception of modular design.

Figure 1 displays a schematic of *FM*, with communication between processes mediated by the communication manager. The figure also shows how file management responsibilities are partitioned between the terminal (5620) and host (UNIX) sides of the programs. Note that all display and manipulation is handled by the terminal via controls (e.g., menus), the keyboard, mouse, bitmaps, messages, and I/O channels. This includes cutting, pasting, and scrolling, as well as other interactive operations in *graphmod* and *dirmod*. The host's responsibilities include providing directory and file information, saving and retrieving graphs and files, non-interactive editing and file commands, and support for such purposes as field verification and help.

Interprocess Communication

Although *graphmod* and *dirmod*, like Pike's *sam* text editor, can be used completely independently for accomplishing their respective tasks, the beauty of FM lies largely in how these distinct programs cooperate in creating a coherent representation of the working environment. This style of interprocess communication extends earlier work by Blewett [2].

The interprocess communication (ipc) module was designed to work in harmony with the Pike graphics process model. Message passing is done asynchronously, which matches the non-blocking, non-preemptive operating system style of the 5620. For example, a process can accurately track mouse gestures while occasionally polling for messages. Of course a synchronous message passing discipline could also be constructed using our system.

If an application is concerned about efficiency, it may request to be scheduled when messages arrive. When the ipc system is used in this event-oriented style, messages are handled as any other 5620 resource (e.g. the keyboard or the receive queue). Pike's model for handling resources is similar in function to the *select* kernel call.

A process may send a message to a specific or to a generic server process. Generic processes are referred to by agreed upon symbolic names. An ipc name service is provided to allow applications to determine which processes are active. These features allow server and user processes to be added dynamically. For example, a user process may wish to spawn a server process if there is not one already active.

The ipc system guarantees that the delivery will be in *fifo* order. Messages may also be selected by the receiving process by tag, generic id, and specific process (a non-forgeable identifier). There is also a facility for monitoring when a message has been processed by the receiving process.

The message data field is a null-terminated linked list of name-value pairs. The complete message is copied when sent. This novel scheme allows for long, multi-command style messages, which proves to be a useful feature, for example, for sending synchronization information when a new server is started. Copying the message allows the sending process to continue without waiting for the message to be received.

It is important for the ipc code to be unobtrusive so that the modular nature of existing programs is not lost. In general, adding message handling to an existing well-structured program

can be accomplished by adding three or four lines of code and a switch to sort through the messages of interest. Each program usually contains an initialization line, send and receive lines, and optional calls to wait for messages to arrive. In processes that already contain resource monitoring calls, the *wait* call is a modification to the existing call.

The File System Graph

Having described our method of interprocess communication, we turn now to the individual programs providing the three levels of representation. At the highest level is *graphmod*, a software tool with a graphical interface for browsing and manipulating directories in a UNIX file system. Running in a window on a user's terminal, it provides an instantaneous picture of all directories (represented by nodes) the user finds relevant at a particular time or for a particular task. By adding and freeing subdirectories, the user can easily specify a picture of any subset of the UNIX directory structure. Such a graph can be saved in an ASCII file for future reference or manipulation, or constructed using the *find* command. Users can also augment a current tree by reading in previously saved graph-files.

The underlying data structure of *graphmod* is a binary tree with a few modifications to facilitate direct manipulation. The terminal and host sides of the program use a common library for manipulating this data structure. Each node has additional pointers so that the structure is both a graph and a linked list. This modification linearizes graph traversal and simplifies design of the interface. Since the graph can be traversed in one direction for drawing and the other for determining which node is under the cursor, *graphmod* can easily handle overlapping nodes.

The idea behind the *graphmod* interface is to give the user control over node selection and positioning in the visual representation. Each node is mouse-sensitive; the node most recently under the cursor is highlighted in reverse video to indicate that further mouse-button clicks in its window will affect it. Pop-up menus provide options that allow a user to vary parameters of the highlighted node, it's children, it's subtree, or it's path. A user can vary the shape and scale of the tree nodes as well as alter their relative positions in the picture. Currently supported visual representations include small squares and dots without labels, and larger rectangles and ellipses that include labels. In either case, the highlighted node is always identified on the banner-line at the top of its window, along with an indication of how many subdirectories it has.

At present, *graphmod* serves only for displaying and navigating through the directory hierarchy, not for changing the file system organization itself. When used in conjunction with *dirmod* below, however, it allows more direct manipulation for performing the actions underlying such directory-oriented commands as *cd*, *pwd*, and *ls*. For example, selecting (clicking) a node in the *graphmod* window marks it, and sends a message to *dirmod* to open the associated directory for browsing and editing. Clicking on a marked node closes that directory.

Directory Forms

Dirmod works at the level below *graphmod*—on individual directories in the particular file system. It allows users to edit file attributes such as permissions and owner, and to execute UNIX commands on specified files that are relevant for file management.

The window-based interface in *dirmod*, derives from the design in Tabs [1,3], but is implemented with bitmap technology in the spirit of *mux* [4,5,6]. Thus, there may be several text frames in a *dirmod* window, each containing a form for editing a directory. Text frames maintain the original source character strings in the terminal, but display them as appropriate bitmap images in arbitrary sized windows.

The form for each open directory corresponds to a collection of controlled input fields (for file attributes) and output areas. Field types are higher-level constructs, not just strings and numbers—for example, filename fields with full wild cards and verification, and even scrollable regions. Moreover, a single field may be used for multiple data types (e.g., alphabetical or numeric for dates).

The text frame created by opening a directory (clicking on the node in the *graphmod* window or by typing its name in a *dirmod* frame) contains a long listing *(ls -agl)* of the files in the associated directory. To add read, write, or execute permission to a file, one simply replaces a dash with an *r*, *w*, or *x* on the associated line (vice versa for removing permissions). Likewise, typing a new name for a file actually performs the intended *mv* operation, while changing the owner or group on the form does the appropriate *chown* and *chgrp*.

Commands can also be entered to execute other UNIX operations on the associated files (e.g., *cp*, *rm*, and *touch*). Opening a file in *dirmod* sends a message to the *sam* text editor to open a text frame for that file, while closing a file closes the associated text frame. Opening or closing a frame from *dirmod* send the appropriate messages to *graphmod*.

Help is available in a separate pop-up window; it appears only when requested, and is tailored to the particular field or context. Verification and completion are also available on request. Movement within or between fields or between windows can be accomplished by emacs- or vi-style commands, as well as by mouse gestures.

Text Editing

Sam [7] is a bitmap graphics editor that can handle multiple files. It provides an integration of the mouse-oriented editing of its predecessor, *jim* [5] with command line editing in the style of *ed*. It can be used for noninteractive, as well as interactive editing. We have chosen to use *sam* in this prototype version because of its compatibility with the other modules, but any other editor would suffice.

The System as a Whole

Figure 2 shows a display of the 5620 screen for the prototype FM system in operation. *Graphmod*, *dirmod*, and *sam* are running in the top, middle, and bottom windows, respectively. Although considerable functionality remains to be implemented, [2] the figure serves to illustrate the user's view of the system.

The *graphmod* window shows the subdirectory structure for FM itself. The top-level directory, */usr1/fm*, is open, as are two of its subdirectories, */usr/fm/edit/sam*, and */usr/fm/dir/forms*. The latter of these is current, as indicated by reverse video and the banner line at the top.

When each directory was marked in *graphmod*, a message was sent to *dirmod* to create an editable form with a long listing of its files (and subdirectories). The top frame in *dirmod* is */usr/fm/dir/forms*, corresponding to the highlighted node in *graphmod*. Similarly, when three header files in */usr/fm/dir/forms* were opened, the filenames were highlighted in *dirmod* and messages were sent to *sam* to provide for their editing. Opening or closing such files in *dirmod* sends appropriate messages to *sam* and vice versa.

Overall, messages are sent whenever an action in one process' window has relevance for others.

2. This includes full bitmap capabilities, such as reshaping, moving, scrolling, and mouse-oriented editing, for individual frames within each process' window.

In this way, FM maintains consistency of representation and establishes an integrated system with cooperating modules.

User interface considerations will play an important role in the modification and enhancement of FM, as well as the development of subsequent modules for providing a manipulable graphic interface for UNIX applications (see [8]). Perhaps the most important of these for FM and its extensions to be appreciated by experienced as well as inexperienced users is that it work in concert with ordinary UNIX interaction. In this regard, we are implementing another program that will provide a shell-like interface for communicating with other processes. For example, when *cd* [3] or *mkdir* is typed, the relevant messages will be sent to the other modules to modify the graphical representations as well as the file system itself. This may help to integrate the advantages of a more graphical approach with the benefits provided by a command language.

3. We wish to thank Andy Koenig for pointing out that the function definition facility of the V9 Shell can be used to arrange for the "cd" command to send a message to the appropriate window to change the directory it displays.

REFERENCES

[1] Blewett, C. D. "Tabs: A Window Based, Extensible, Highly Typed, Electronic Forms Package," ***Unpublished manuscript,*** 1983.

[2] Blewett, C. D., Freed, A., Hilbert, R. J., Langer, J., Mascitti, R. J., Rodine,C. R. and Weber, W. P. "The Aegis System," ***USENIX Computer Graphics Workshop,*** 1985.

[3] Blewett, C. D. and Hicks, K. I. "Tabs 2.0 Manual: Tools for Creating Window Based Electronic Forms," ***Unpublished manuscript,*** 1985.

[4] Pike, R. "Graphics in Overlapping Bitmap Layers," ***Trans. on Graphics*** **2** (2), pp. 135-160, Reprinted in Proc. SIGGRAPH 1983, 331-356.

[5] Pike, R. "The Blit: A Multiplexed Graphics Terminal," ***Bell Labs Tech. J.*** **63, #8, part 2,** pp. 1607-1631, Oct. 1984.

[6] Pike, R. "mux: A Window System with a Simple User Interface," ***Unpublished manuscript,*** 1986.

[7] Pike, R. "Structural Regular Expressions," ***Unpublished manuscript,*** 1986.

[8] Wish, M., Helfman, J. I., Edmark, J. T., Blewett, C. D., "Taming the UNIX® Interface: A New Approach and Prototype," ***Unpublished manuscript,*** 1986.

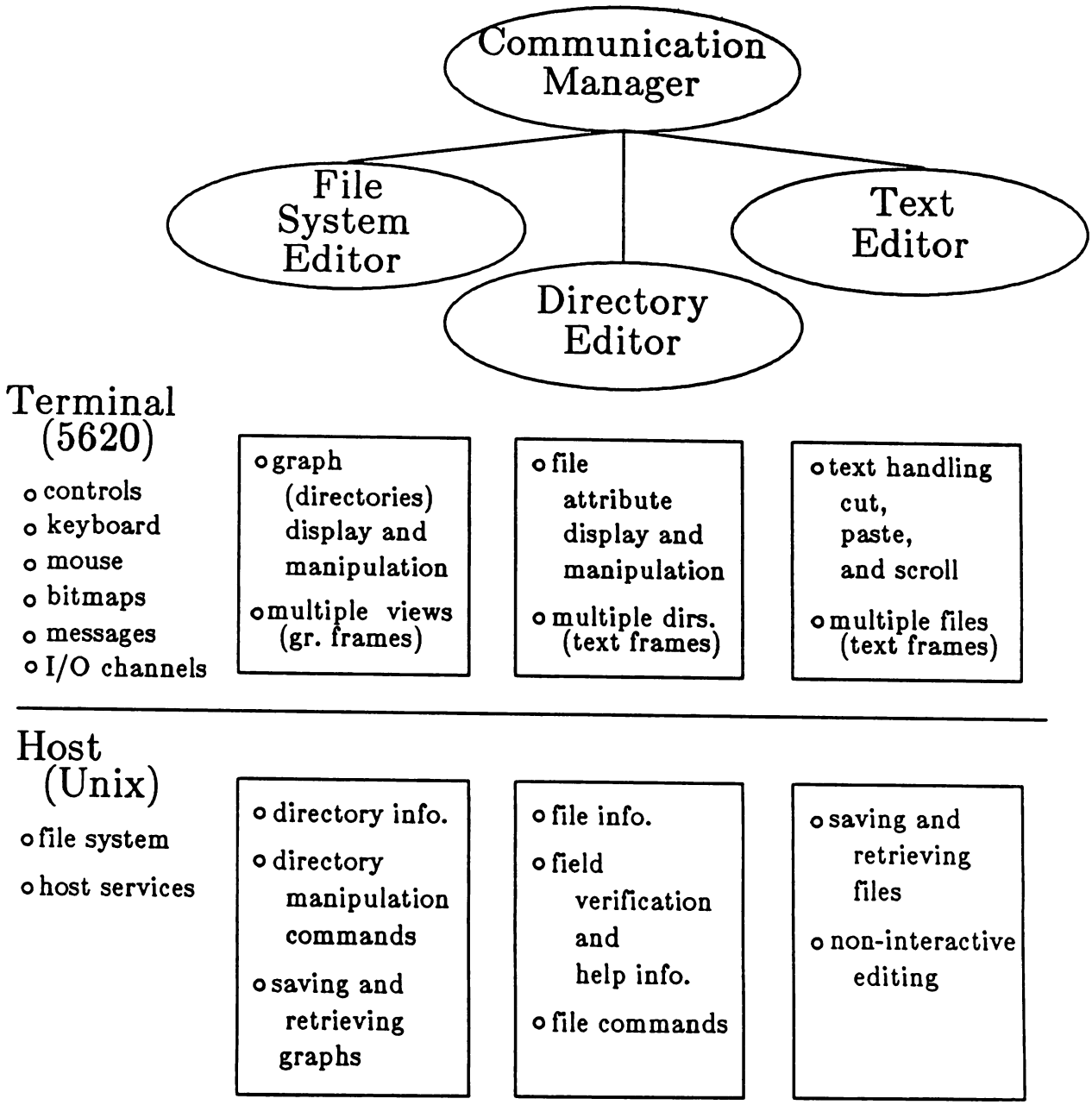


Figure 1.

Schematic of *FM*, with communication between the file system editor, *graphmod*, the directory editor, *dirmod*, and the text editor, *sam* mediated by the communication manager. The terminal (5620) side of each program is responsibile for display and manipulation via controls, the keyboard, mouse, bitmaps, messages, and I/O channels. The host provides file system information and other UNIX services.

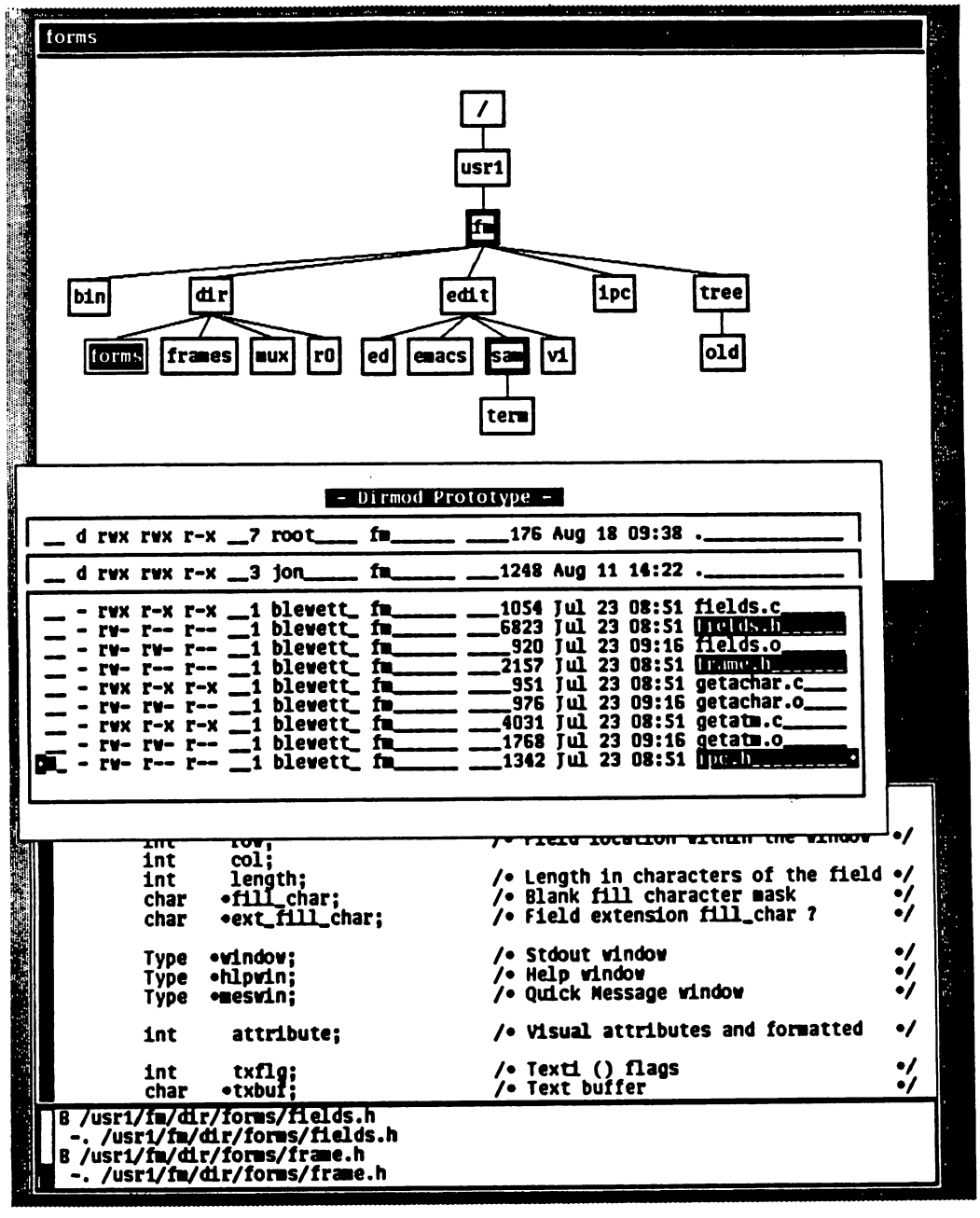


Figure 2.

Screen display of prototype *FM* system in operation. Graphmod, which is running in the top window, shows a graph of the directory structure of *FM* itself. As indicated by double borders, three directories are open, */usr1/fm*, */usr1/fm/edit/sam*, and */usr1/fm/dir/forms*. The last of these is current, as indicated by highlighting and the banner line on top. The three open directories are displayed in the Dirmod window, with */usr1/fm/dir/forms* on top. Three header files in this directory are open, *fields.h*, *frame.h*, and *ipc.h*. The bottom window shows the command window and an open file in *sam*.

Scattered Thoughts on B-Splines

Spencer W. Thomas
Computer Science Dept.
University of Utah
Salt Lake City, Utah 84112
USA

November, 1986

Abstract

B-splines are widely used in computer graphics and related fields. They are also widely misunderstood or only partially understood. To many people the term "B-spline" means a particular, narrowly defined, curve or surface type. Many problems arising from this narrow understanding could be averted by understanding and using the full B-spline definition. We attempt to provide a small push in that direction through examination of some examples of poor application of B-splines.

1. Introduction

Folklore concerning the use of B-splines in a number of applications has been passed on from one academic generation to another. Much of this folklore, while perhaps not incorrect, is certainly erroneous, in the sense that better techniques exist to achieve the same goals. In this paper we explore some of this folklore and its (mis)application.

We start with a quick overview of B-splines. The main body of the paper is a discussion of several examples. We conclude with a brief view of the B-spline as a "black box", the inner workings of which we can ignore.

2. B-splines

There are many ways to define a B-spline (for more information see, e.g., [5, 11, 10]). I like to start from the abstract and work towards the concrete. A *spline* curve is a piecewise curve with a specified continuity constraint at the *knots* between the pieces (or *spans*). In other words, it is composed of some number of well-defined curve spans connected together. The points where the spans are joined are called knots. At the knot where one span joins the next, the position and some number of derivatives of each curve are identical. We speak of *degree* of continuity, meaning the number of derivatives that are continuous from one side of the knot to the other. The position and first and second derivatives of a spline with C^2 continuity are continuous along its entire length.

Although there are exceptions, most splines are piecewise polynomials, probably because these are easy to compute. The n^{th} derivative of a polynomial of degree n is constant, so if we join two such polynomials together with C^n continuity, they must be identical. This produces an uninteresting spline. It is possible to join different polynomials of degree n together with degree $n-1$ continuity, so a spline whose pieces are n^{th} degree polynomials will usually have continuity in the first $n-1$ derivatives.

2.1. Parametrization and Basis Functions

Well, this is mildly interesting, but where do B-splines come in? I'm getting there, but first we need a couple of other concepts. The first is related to the *parametrization* of the curve. In computer graphics we generally represent a curve in a parametric form, that is, each of the coordinates at a point on the curve is determined as a function of a single parameter t. As t varies from its minimum value to its maximum value the curve is traced out. In the case of a spline, at certain values of t we will move from one span to the next (this occurs, you recall, at the knots). The ordered set of the parameter values corresponding to the knots is called the *knot vector*.

Since a spline is a pieced-together curve, we can really only write down nice equations for each piece separately. Thus, we must say that from t_0 to t_1, the curve is represented by this polynomial, and from t_1 to t_2 by that polynomial, and so on. This gets tedious quickly, and hides any unity that might exist between the spans. Instead, we often write the curve in terms of *control points* and *basis functions*. While polynomial coefficients have very little intuitive relationship to the shape of the curve, the control points typically have geometric meaning. The basis functions are themselves splines, but of a restricted sort (exactly what sort depends on the type of spline), and are presumably easier to calculate than a general spline. Essentially what we have done is to hide the piecewise nature of the spline under the basis functions.

A curve (any curve, not just a spline) represented in this way can be written as

$$S(t) = \sum_i P_i B_i(t),$$

where the points P_i are the control points and the functions $B_i(t)$ are the basis functions. Another simplification that we can note here is that the basis functions are normal scalar-valued functions, all the 2-D or 3-D nature of the curve comes from the control points. The control points are often drawn with line segments connecting successive points. The resulting figure is called a *control polygon.*

2.2. Curve Properties

Having defined a curve in terms of some geometric control points, there are some properties we would like it to satisfy. The most basic of these is that the curve not change shape if the set of control points is moved rigidly (or if the coordinate system is moved under them). This will be true if the basis functions sum to one for all parameter values. If the basis functions are also always non-negative, then the curve will satisfy the *convex hull* property. That is, the curve will always lie inside the convex hull of the control points. This is a desirable property, as it allows us to determine the potential extent of the curve for purposes such as clipping, and makes the curve shape more predictable to the user.

2.3. B-splines

"Where are the B-splines?" you ask. Well, the "B" in B-spline comes from the word "basis"[1]. The B-spline basis functions are special splines. They have a property called *minimal local support.* Local support means that outside some finite range of parameter values, the function is zero. Minimal local support means that you can't make this range any shorter without forcing the function to be zero everywhere. Consider a piecewise linear spline (you may prefer to think of it as a "polyline", in computer graphics terms). Since we require that the zeroth derivative (i.e., the position) of the spline is continuous, it must be non-zero over at least two spans. In one span it can go away from zero, but it can't change direction and come back to zero until the next span. A polynomial spline of degree n must be non-zero over $n+1$ spans to have C^{n-1} continuity. We often refer to the *order* of a B-spline, this is simply the degree plus one, and is the number of spans over which a basis function is non-zero.

So, given a knot vector, we can define a set of B-spline basis functions over that knot vector. Each basis function will "start" (move away from zero) at a different knot value. Two spline functions with minimal local support that start at the same knot must be multiples of each other (I am not going to prove this here). We *normalize* the basis functions so that they are non-negative sum to 1 everywhere (it is possible to do this, but I will not prove that, either). Finally, we can use the basis functions to blend a set of control points and thus define a curve.

As you may have noted from the construction of the basis, B-spline curves satisfy both the properties described above. In addition they have two other useful properties. *Local control* means that if you change the value of a control point, only a region of the curve near the control point will

[1] And you thought it was someone's initial.

be affected. This is a direct consequence of the local support of the B-spline basis functions. Outside the range where the corresponding basis function is non-zero, a given control point can not affect the curve at all. B-splines are also *variation diminishing*. This means that any straight line will intersect the control polygon at least as many times as it intersects the curve. The only "wiggles" in the curve are those that were explicitly put there.

There is one other complication I have neglected to mention. Sometimes we want to put discontinuities of one sort or another into the curve. Well, if you go through the mathematical derivation of the B-spline basis functions, you will find that a "multiple" knot value will lower the continuity degree of the spline at that knot. A multiple knot is simply one that appears more than once in succession in the knot vector. Each repetition of the knot value lowers the continuity degree at that knot by one. For a B-spline of degree n, a knot that is repeated n times eliminates all derivative continuity, leaving only continuity of position. Repeating the knot value one more time lets the curve be completely discontinuous at that knot. This is not usually done, except at the ends of the curve.

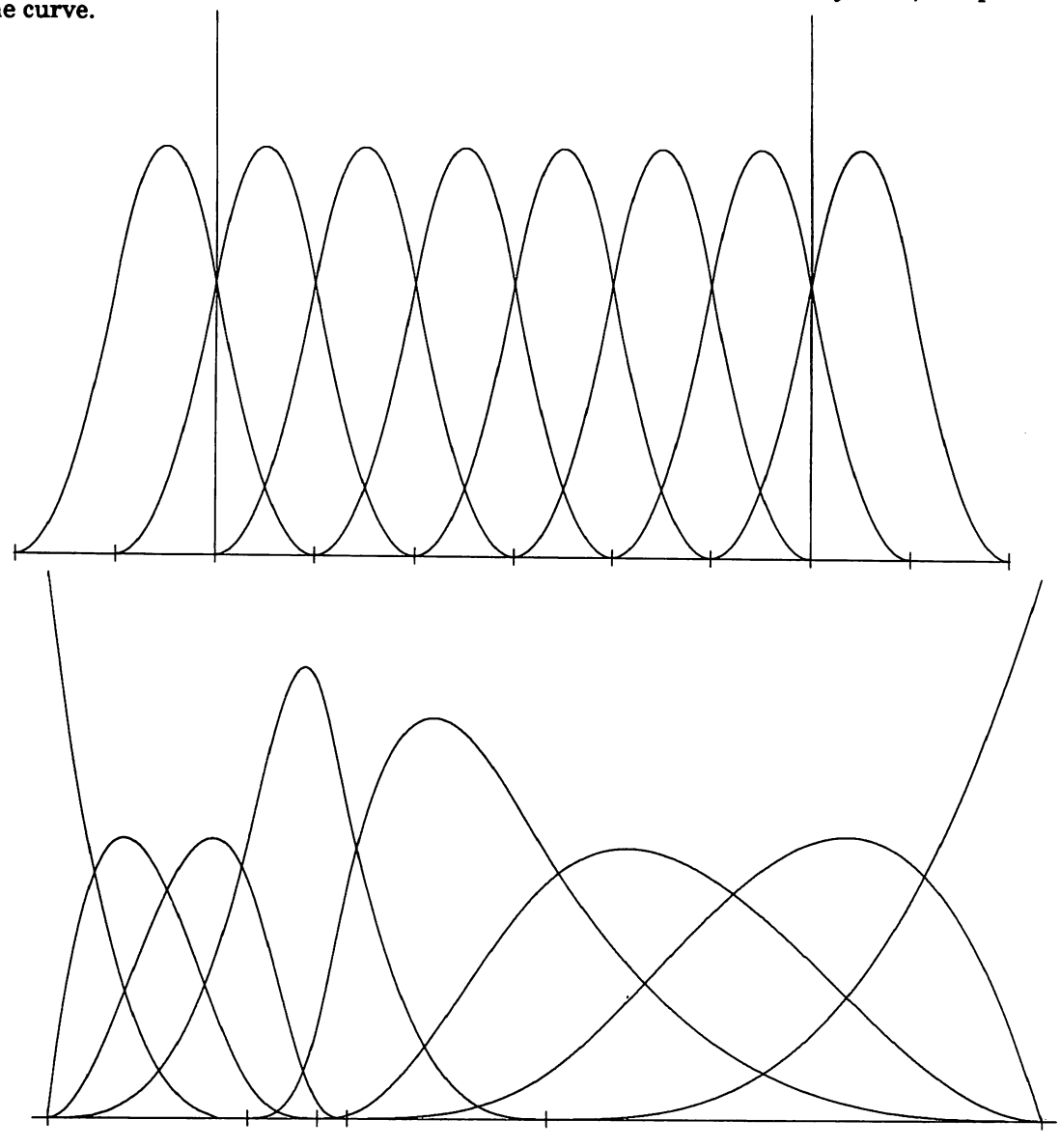


Figure 1. Some B-spline basis functions

2.4. B-spline Basis Functions

Let's look at some basis functions (Figure 1). At the top are some *uniform* basis functions for a quadratic B-spline. The knot values (indicated by tick marks) are evenly spaced, so all the basis functions look the same. This makes them easy to compute; a uniform basis is frequently used for this reason. A curve built from these basis functions is only well defined where the basis functions sum to one; this is the region between the vertical lines in the figure. Although they apparently are not used, it is necessary to define knot values outside the range of the curve in order to completely specify the basis functions.

Consider a curve defined using the quadratic uniform basis functions. It starts at the parameter value of the first vertical line. The initial point of the curve lies exactly halfway between the first and second control points. This is a problem with the uniform basis functions: curves built with them do not interpolate the initial and final control points. We call this a *floating* end condition. A B-spline curve will almost never (except by accident or because of a multiple knot) interpolate a control point other than the first and last (this is implied by the convex hull property and by the continuity of B-splines), but we often would like it to at least interpolate the endpoints. This desire has led to some disgusting "hacks" (see below).

Placing a knot of multiplicity equal to the order at each end of the parameter range of the curve eliminates the need to have knots outside this range, and corresponds to the so-called *open* end conditions. The bottom of Figure 1 shows cubic B-spline basis functions defined over a non-uniform knot vector with open end conditions. The basis functions are all different, but they still sum to one, have local support, and are continuous in the second derivative (except at the end points).

2.5. Manipulating B-splines

So far, it would seem that to evaluate a B-spline, we need to figure out the basis functions. And this seems to be non-trivial (to put it mildly) in any but the uniform case. There is some validity to this view. We can ignore the problem, though, because it is possible to evaluate a B-spline without ever figuring out a single basis function. The basic algorithm for evaluating a B-spline at a particular parameter value was developed by deBoor and Cox. It is a recursive algorithm involving the knot vector and the control points. See deBoor [5] for the details.

Two other common operations on B-splines are *refinement* and *subdivision*. It is possible to add new knot values to the knot vector of a B-spline and figure out new control points (corresponding to the basis functions defined on the new knot vector) so that the resulting curve is identical to the original. Well, almost identical. The new curve has some *pseudo-knots*. These are knots which join two spans of the same polynomial curve, and occur where the new knot values were inserted. If we change the value of some control points, the pseudo-knots can become "real" knots. It turns out that the new control points are always closer to the curve than the old ones were, and that as more and more knots are added, the control polygon converges to the curve.

Subdivision is a special case of refinement. To subdivide a curve, you insert a knot of multiplicity equal to the order of the curve (four-fold for a cubic B-spline). This creates a potential positional discontinuity in the curve, essentially dividing it into two separate curves. The two curves taken together make up the original.

A curve can be evaluated at a parameter value by inserting a knot of multiplicity equal to the degree of the curve (a triple knot in a cubic curve). One of the new control points resulting from refinement lies on the curve.

There are several algorithms for refinement and subdivision. Cohen, Lyche, and Riesenfeld [4] developed the *Oslo algorithm*. With the Oslo algorithm, any number of new knots can be inserted into a B-spline simultaneously. Boehm [2] developed an algorithm for inserting a single knot into a B-spline. This is essentially a special case of the Oslo algorithm, but is more efficient in the case of inserting only a few knots. Lyche and Moerken have since improved the efficiency of the Oslo algorithm [8] so that it does the minimum amount of computation in all cases.

2.6. B-spline Surfaces

B-splines are not restricted to defining curves, we can also create B-spline surfaces. The only commonly used form is the *tensor product* B-spline surface. Mathematically, it is defined by an array or *mesh* of control points (instead of the curve's control polygon) and two sets of basis functions, one set for each of the two parametric directions. We write

$$S(u,v) = \sum_i \sum_j P_{ij} B_i(u) D_j(v)$$

Generally the two parametric directions will have different knot vectors, and therefore different sets of basis functions. They may also have different orders (e.g., a cylindrical surface might be best defined as a surface that is quadratic in one direction, but linear in the other).

2.7. Rational Splines

A *rational* spline is build of basis functions that are piecewise rational polynomials. In other words, each basis function is the ratio of two polynomials. We write

$$S(t) = \left(\frac{x(t)}{w(t)}, \frac{y(t)}{w(t)}, \frac{z(t)}{w(t)}\right)$$

The polynomial $w(t)$ is called the *weight*, and can be represented as a B-spline, just as the functions $x(t)$, $y(t)$, $z(t)$ are. We let the control points be *homogeneous* points,

$$P_i = (x_i w_i, y_i w_i, z_i w_i, w_i).$$

Varying the control point weights changes the shape of the curve. Rational curves are most useful for exactly representing conic sections (see below).

3. Examples

This section contains a number of unrelated examples tied together by a common thread. In most of them, an alternative to a sub-optimal, but common method of using B-splines is demonstrated. In most of these examples, the problem arises from the reluctance or refusal to use the full B-spline representation, but instead to remain tied to a particular subdomain.

The first two examples describe problems arising from the use of uniform basis functions for computing B-splines. A uniform basis is easy to use because all the basis functions are identical. They can be precomputed and tabulated, or loaded into a matrix multiplier unit for quick computation up to cubic, or converted to polynomials and evaluated using forward differencing. A curve is then defined completely by specifying its control polygon. No need to worry about parametric range, knot vectors, lots of different basis functions, and so on.

However, insistence on a uniform basis is the underlying cause of a number of common faults in B-spline usage. The uniform B-spline, while easy to compute, is really not a very "nice" curve. You have to go through contortions to get it do some seemingly simple things. A couple of these are discussed below.

3.1. End Conditions

People using B-splines who like to use a uniform basis with floating end conditions, they then have to deal with the problem that the curve does not usually come anywhere near the endpoints of the control polygon (or corners and edges of the control mesh). The insistence on maintaining the uniform basis knot vector has led to some Baroque schemes for achieving endpoint interpolation.

The most common of these, used with cubic splines, is to repeat the initial and final vertices of the control polygon. Repeating a terminal vertex once (giving a "double vertex") forces the curve to end on the edge between the terminal vertex and the adjacent vertex, close to the terminal vertex. With a little arithmetic, this can be hidden from the user by simply moving the terminal vertex 1/6 of the way towards the adjacent vertex before drawing it.

Let's look at what happens here, though. The tangent vector at the endpoint is proportional to the

difference of the first and third vertices. The second derivative of a uniform, floating B-spline at an endpoint is a linear combination of the three terminal control points. Since these three points all lie on a straight line, the second derivative is parallel to the tangent. This means that the curve has a "flat" spot at the end.

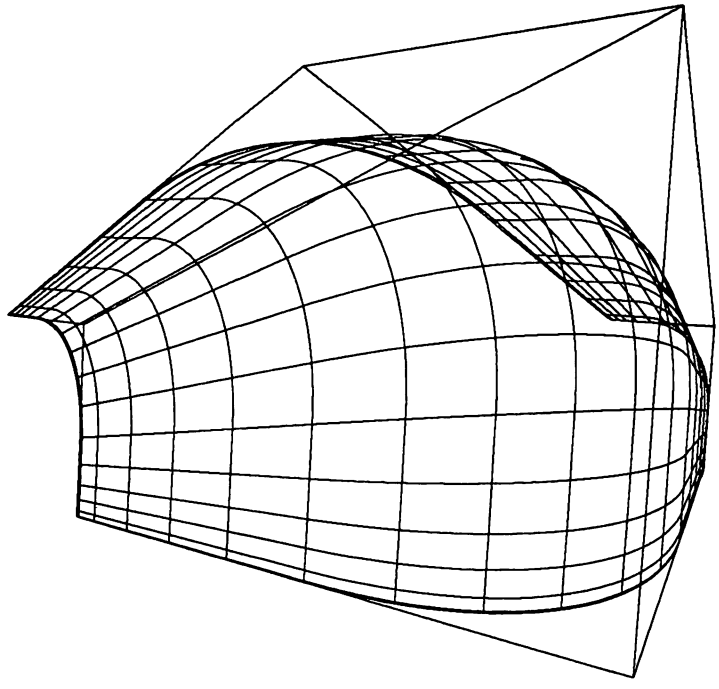


Figure 2. Surface with triple vertices, showing flat regions.

If we miss the "trick" used above to make the curve appear to interpolate a double vertex, we may be tempted to triple the vertex to force interpolation. This was recommended in a thesis by Sweeney [13]. In this situation we find that the first curve segment, which depends on the first four control points, is a straight line segment! Even if we could live with the point of zero curvature introduced by the double vertex, this is too much. In the surface case, you get a bilinear "ribbon" around the whole surface (Figure 2). The flat regions at the corners of this surface are particularly noticeable.

A "correct" solution, one that provides endpoint interpolation with full flexibility at the endpoint (no zero tangent, no flat spots) is to use, not a multiple vertex, but a multiple knot. In the cubic case, the use of a quadruple knot at the ends of the span produces interpolation at only a slight cost in increased complexity. The cubic basis functions for this *uniform, open* case are shown in Figure 3, together with a spline using them. If the number of vertices is equal to the order (4 for cubic), the resulting curve is a Bezier curve.

3.2. Discontinuities

Occasionally, one wishes to put a discontinuity in the middle of a B-spline curve. For example, the "face" profile shown on the left of Figure 4 has a number of tangent discontinuities. Again, the common solution is to introduce a multiple vertex. For a quadratic curve, such as the face profile, a double vertex is required to achieve tangent discontinuity. However, the curve on either side of the double vertex will lie exactly on the control polygon for half the length of the adjacent edges. An example of this behavior is shown on the right of Figure 4, where multiple vertices have been used at the points of tangent discontinuity.

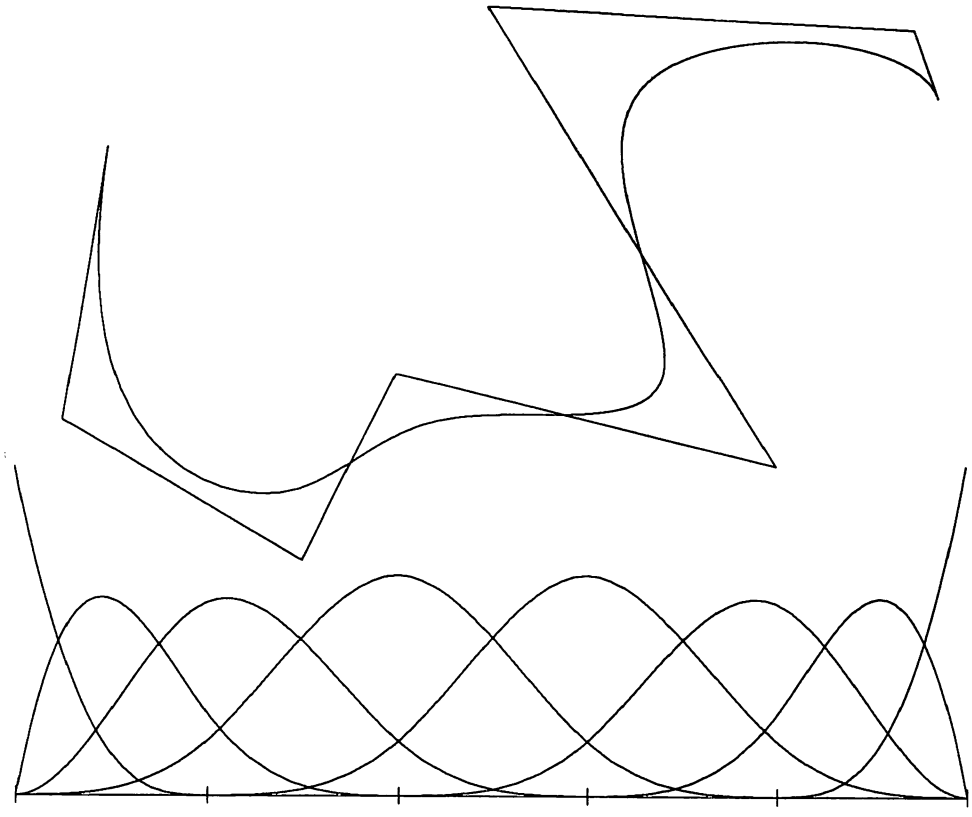


Figure 3. Uniform, open cubic basis functions and curve.

Again, a better solution is to use the full power of the B-spline representation and introduce a multiple knot at the point where a discontinuity is desired. Each knot reduces the continuity degree by one. Thus, when starting with a cubic curve, which has C^2 continuity normally, a double knot will allow a discontinuity of the second derivative (curvature), and a triple knot will allow a tangent discontinuity to be introduced. A quadruple knot provides for a discontinuity of position. The cubic surface shown in Figure 5 has a quadruple knot line running from top to bottom in the middle. The control mesh has been pulled apart to introduce a hole into the surface. Note that although the control mesh is connected, the resulting surface is not.

If open end conditions have been implemented, introducing tangent discontinuities into a curve requires no extra basis function computation. A tangent discontinuity occurs at a knot of multiplicity *order–1*. The basis functions near this knot are identical to those at the ends of an open spline, with one exception. The only difference is in the basis function corresponding to the vertex being interpolated. At an endpoint (*order* multiplicity), it is one-sided, dropping abruptly to zero on the other side of the knot. At a tangent discontinuity, it is symmetric; each side has the same shape as an endpoint basis function.

3.3. Adding Flexibility

Knapp, in his PhD thesis [7], describes a method of adding flexibility to a region of a B-spline curve or surface by inserting knots. With the new knots come more curve segments and control vertices. The new vertices can then be moved to achieve a localized change in the shape of the spline. He did this by evaluating the curve at the desired new knot values, and then interpolating the knot points with a spline based on the new knot vector. The new spline was exactly the same shape as the old one, but had more knots. Unless new knots are inserted uniformly throughout the entire length of the curve, the resulting knot vector will be non-uniform. Interpolating the new

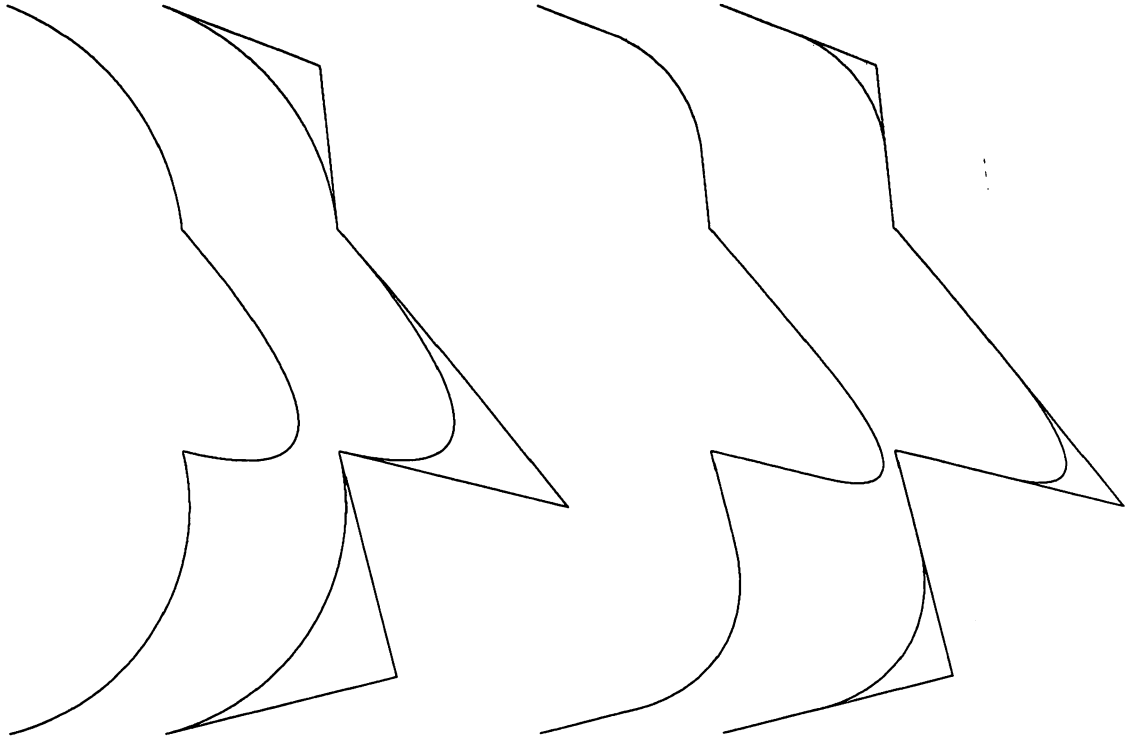


Figure 4. Profiles with multiple knots and multiple vertices.

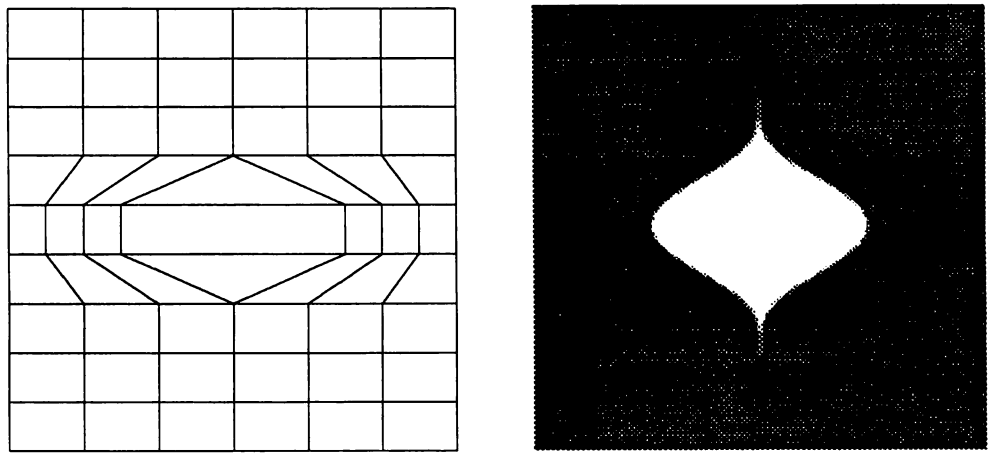


Figure 5. Hole in a surface with multiple knots

points with a uniform knot vector will therefore result in a different curve.

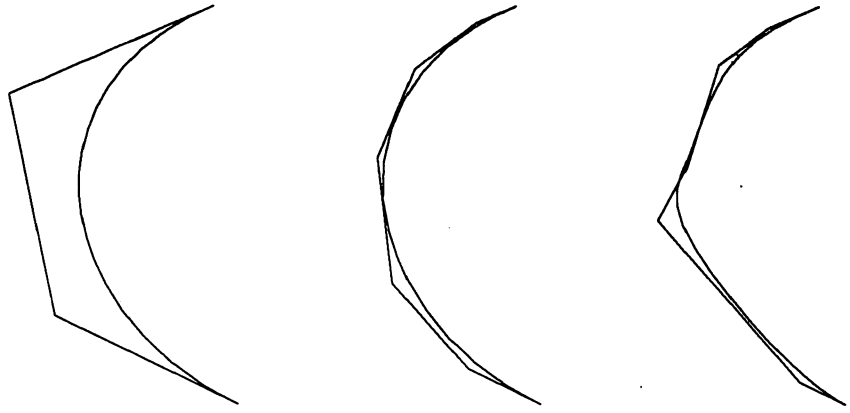


Figure 6. Adding flexibility to a curve

An example of adding flexibility is shown in Figure 6. The original curve and control polygon are shown at the left, the result of adding some new knots is shown in the middle, and the (different) curve resulting from interpolation of the new knot points with a uniformly parametrized spline is shown at the right.

Although the technique works, it rapidly becomes unwieldy as the size of the curve or surface increases, and is subject to numerical instability in the interpolation process. A knot insertion algorithm, such as the Oslo Algorithm, can perform this task directly, more quickly, and more accurately than Knapp's technique. To be fair, at the time he did his work, knot insertion algorithms were unknown. There is no excuse to use it today, though.

3.4. Animation control

In a paper at the 1985 SIGGRAPH conference, Steketee described an animation motion control method using spline paths [12]. He started with "key" positions and interpolated them with a B-spline. He manipulated the shape of the curve using the B-spline control vertices. The local control property of B-splines meant that it was easy to change one section of motion without affecting other sections.

Where the method fell apart was when he wanted to join two independently created motion paths. Of course, the concatenated paths were generally not smoothly joined. To create a smooth joint, he abandoned the B-spline and created a new interpolant through the combined set of keyframes. Naturally, this usually introduced wiggles near the "joint" between the two paths. So he modified the position of the control points near the joint with a "phrasing" heuristic to reduce the wiggle. In the process, interpolation of the key points near the joint was lost. The interpolation process changed the shape of the curve everywhere, and this was not corrected by "phrasing" the joint.

This is a prime example of not taking full advantage of the B-spline representation. Assuming the original curves were cubic splines, they can be joined by discarding one copy of the common control point at the joint, and concatenating the control polygons and knot vectors with a triple knot at the joint (see Figure 7a). Now we have a single path, but with a tangent discontinuity at the joint. We can smooth the discontinuity, at the cost of losing interpolation at the joint (which would have happened anyway), by "spreading out" the triple knot. As the knots are moved further apart, the corner at the joint becomes less abrupt. Not only does this provide control over the amount of

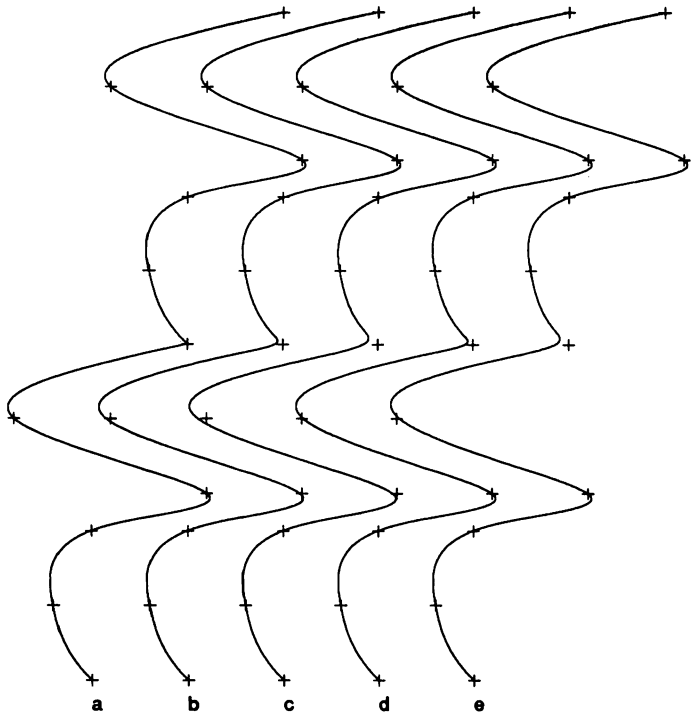


Figure 7. Joining curves smoothly

smoothing, it avoids a costly interpolation calculation that would affect the shape of the curve over its entire length. In the figure, b and c show the curve with two different knot spreads. Note that the two points near the joint are no longer interpolated exactly. We can further isolate the effect of the change by inserting an extra knot on either side of the joint, outside of the "spread" interval. The effect of doing this is shown in d and e (where the spread values are the same as in b and c, respectively).

3.5. Splines with tension

Recently, attention has been focused on various tensioned spline formulations [1, 9]. Tensioned splines provide the user with more control over the shape of the curve by providing one or more "tension" parameters at each control vertex. The parameters may be used to pull the spline closer to the control point, or to skew it to one side or another. Since these effects rely on introducing carefully controlled discontinuities at the knots, they seem difficult to achieve with B-splines.

In actuality, there are at least two ways to accomplish similar effects with a B-spline. It is possible to "skew" the curve by changing its parametrization. If the parameter range following a knot is shortened, the curve will be skewed towards that side of the knot. An example is shown at the top of Figure 8[2], in which the parametric range following the third knot was multiplied by 8.

[2]Compare Figure 3 of [6].

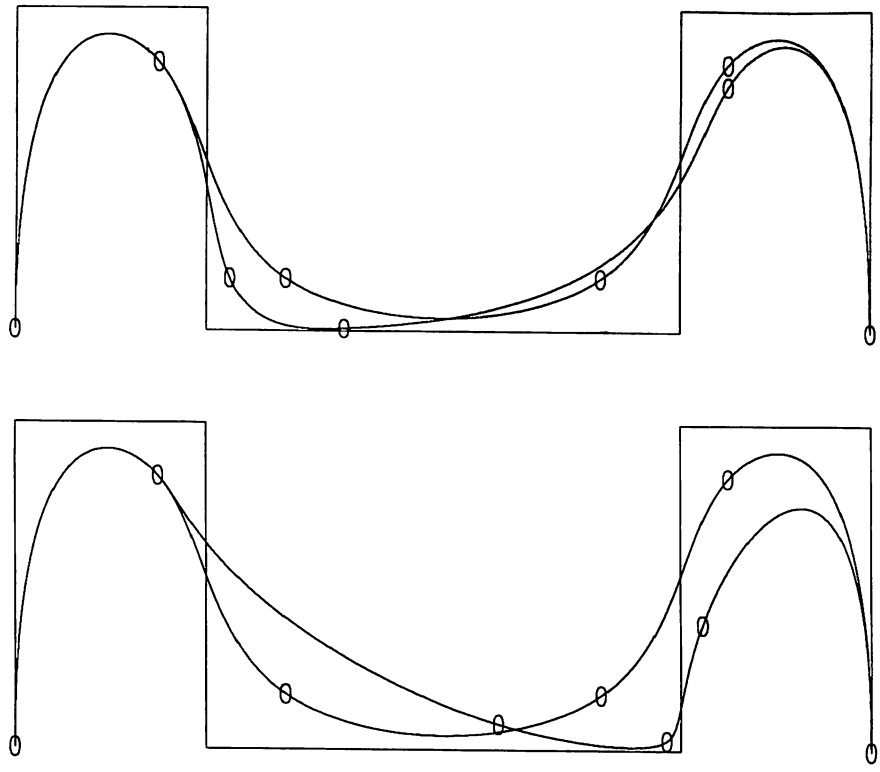


Figure 8. Simulating tension with a B-spline

The amount of skew is related to the ratio of the lengths of the parameter intervals. Since an affine transformation of a knot vector leaves the curve unchanged, the skew effects may be localized to one region of the curve[3]. An effect identical to that of the β_1 parameter of the beta-spline is produced this way.

The curve may be "pulled" closer to or "pushed" away from a control point by using a rational curve formulation. Increasing the weight at a control point will cause the curve to come closer to that point; decreasing the weight causes the control point to have less effect on the shape of the curve. The bottom of Figure 8 shows a simple example, where the weight of the lower right point in the "U" was set to 8. Unfortunately, this does not work as well. The modified control point pulls "too hard" on the curve.

Cohen [3] has introduced tensioned B-splines, a method of reproducing beta-splines exactly by building on top of the B-spline formulation. By introducing (into a cubic curve) a triple knot at each knot location and controlling the positioning of the newly introduced control points, the effect of the beta-spline parameters can be achieved. The user need never see the extra knots or control points, but instead is provided with a B-spline-like curve with tension and skew control. This curve still has all the B-spline properties: local control, convex hull property, variation diminishing property (the last two in relation to the underlying B-spline control polygon), and can be refined and subdivided. The large body of theory and techniques developed for B-splines is directly applicable to these tensioned splines.

[3]Basically, instead of scaling just the interval following the knot, the entire parameter range following the knot is scaled.

3.6. Drawing B-splines

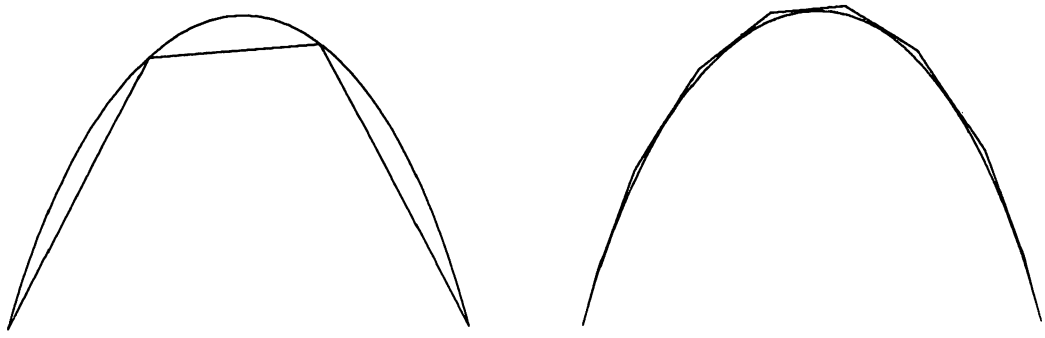


Figure 9. Two methods of drawing a B-spline

The traditional method of drawing a spline is to evaluate points on the curve and connect them with straight lines. Particularly when using non-uniform splines, this requires a fair amount of computation per point. Evaluating a B-spline at a point is equivalent to inserting a knot with multiplicity equal to the order of the spline (4 for cubics). For the same amount of computation, we could insert 4 different knots and get a *refinement* of the curve. The refined control polygon will often be a better approximation to the curve than the usual evaluation method produces. Figure 9 illustrates a comparison between curve evaluation and refinement as a strategy for drawing a B-spline curve. Both approximations required about the same amount of computation to produce[4]; the refinement method clearly yields a better approximation.

4. Special cases

Some particular special cases deserve our attention because of either their simplicity or their recurrence in applications. For those who insist on computing basis functions instead of using a refinement method to draw splines, there are knot vectors that yield simple basis functions, but that provide most of the advantages of a non-uniform spline.

The uniform, open end condition spline, mentioned above, frequently provides a good starting point when designing a shape using a spline. In *ab initio* design, we are exploring different shape possibilities, and don't want to be burdened with extra complexity. Once we get close to the desired final shape, we may then "tune" the parameterization. In some applications, the use of open end conditions in place of the usual floating end conditions is sufficient, and avoids the need for "hacks" to force endpoint interpolation.

Internal tangent discontinuities can also be introduced without requiring any new basis functions. The basis functions in the vicinity of the multiple knot required for a tangent discontinuity are identical to those at the ends of an open spline.

Another useful special case, particularly in drafting-like applications is the "piecewise Bezier" knot vector. It has an *order*-fold knot at each end and equally spaced *order–1*-fold knots in the interior. This produces a curve that consists of Bezier curves connected at their endpoints. A quadratic rational piecewise Bezier spline is a good representation for a profile made up of connected straight lines and circular arcs.

[4] 7.5ms for the left image, 7.9ms for the right one.

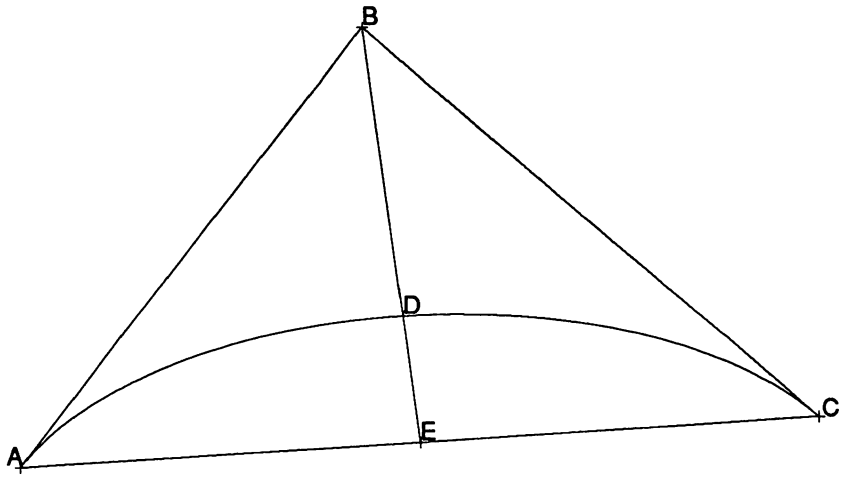


Figure 10. Conic section curve definition

Conic curves are frequently used in mechanical design. They may be represented exactly by quadratic rational piecewise Bezier curves. The standard "triangle" method of specifying a conic is easily adapted to computing the appropriate control points for a Bezier conic curve. Refer to Figure 10. The points **A** and **C** are the first and last points of the control polygon. They each have a weight of *1.0*. The point **B** at the vertex of the triangle is the Euclidean projection of the middle control point. Let λ=**DE**/**BE**. Set the weight w for **B**: $w=\lambda/1-\lambda$. The resulting Bezier curve exactly reproduces the conic curve.

A conic curve subtending more than 180 degrees can be created by using a negative weight, but is more commonly represented by a piecewise Bezier curve, each segment of which is smaller than 180 degrees.

5. B-splines as a Black Box

Knot insertion algorithms for refinement and subdivision of B-splines permit us to view the spline as a "black box". That is, we need not be concerned about the underlying polynomial representation, but can view a B-spline as a curve that is defined by a knot vector and set of control points. Manipulating these handles modifies the curve in well-defined ways, and inserting knots brings the control points closer to the curve, increases its flexibility, and lets us create discontinuities in it. The last bastion of basis functions, curve drawing, is wiped out by using refinement to draw it.

Once freed from dependence on the piecewise polynomial nature of the spline, we realize that our concentration on the trees kept us from seeing the forest. We felt that we knew the best ways to deal with polynomials, and seeing the spline as a collection of "friendly" curves prevented us from considering it as an entity in its own right.

6. Acknowledgements

I would like to thank all those I have worked with and learned from at Utah, where I gained my working knowledge of and experience with B-splines. I am also grateful to those whose bad examples inspired this work.

This work was supported in part by the National Science Foundation (DCR-8506115 and DCR-8121750), and the Defense Advanced Research Projects Agency (DAAK11-84-K-0017). All opinions, findings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Barsky, B. A.
The Beta-spline: A Local Representation Based on Shape Parameters and Fundamental Geometric Measures.
PhD thesis, University of Utah, December, 1981.

[2] Boehm, W.
Inserting New Knots into B-spline Curves.
Computer-Aided Design 12(4):199-201, July, 1980.

[3] Cohen, Elaine.
A New Local Basis for Designing with Tension Splines.
Technical Report UUCS-85-104, University of Utah, June, 1985.

[4] Cohen, E.; Lyche, T.; and Riesenfeld, R. F.
Discrete B-splines and Subdivision Techniques in Computer-Aided Geometric Design and Computer Graphics.
Computer Graphics and Image Processing 14(2):87-111, October, 1980.
Also Tech. Report No. UUCS-79-117, University of Utah Computer Science Dept, October 1979.

[5] deBoor, Carl.
Applied Mathematical Sciences. Volume 27: *A Practical Guide to Splines.*
Springer-Verlag, New York, 1978.

[6] Goodman, T. and Unsworth K.
Manipulating Shape and Producing Geometric Continuity in beta-Spline Curves.
IEEE Computer Graphics and Applications 6(2):50-56, February, 1986.

[7] Knapp, L. C.
A Design Scheme Using Coons Surfaces with Nonuniform B-spline Curves.
PhD thesis, Syracuse University, September, 1979.

[8] Lyche, T. and Moerken, K.
Making the Oslo Algorithm More Efficient.
SIAM J. Numer. Anal. , to appear, 1984.
Also available as Research Report in Informatics No. 87, University of Oslo (August 1984).

[9] Nielson, Gregory.
Rectangular nu-Splines.
IEEE Computer Graphics and Applications 6(2):35-40, February, 1986.

[10] Riesenfeld, Richard F. and Gordon, William J.
B-spline Curves and Surfaces.
Computer Aided Geometric Design.
Academic Press, 1974, pages 95-126.
eds. Barnhill and Riesenfeld.

[11] Riesenfeld, R. F.
Applications of B-spline Approximation to Geometric Problems of Computer-Aided Design.
PhD thesis, Syracuse University, May, 1973.
Available as Tech. Report No. UTEC-CSc-73-126, University of Utah Computer Science Dept.

[12] Steketee, Scott and Badler, Norman.
Parametric Keyframe Interpolation Incorporating Kinetic Adjustment and Phrasing Control.
In Brian A. Barsky (editor), *Proceedings of SIGGRAPH '85*, pages 255-262. ACM SIGGRAPH, August, 1985.

[13] Sweeney, Michael and Bartels, Richard.
Ray Tracing Free-Form B-Splien Surfaces.
IEEE Computer Graphics & Applications 6(2):41-49, Feb, 1986.

PROCEDURAL SPLINE INTERPOLATION IN UNICUBIX

Carlo H. Séquin

Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

ABSTRACT

UNICUBIX is an extension to the Berkeley UNIGRAFIX modeling and rendering system.[1] The original system, restricted to polyhedral object, is extended by permitting cubic curves in place of the previous linear edges between pairs of vertices, and triangular or quadrilateral patches between such curved borders. Where desired, the patches are joined together with G^1-continuity using procedures that guarantee locality of control.

A sequence of procedural steps determines first the vertex normals, then the Bézier control points for the curved edges, and finally the internal control points for all patches. These steps use an intuitive geometric approach to fitting spline curves and surfaces through the given set of vertices. A variety of rules and procedures makes it possible to provide pleasing default values for even difficult situations. There are no restrictions on the topology of the original polyhedral net defining the shape of the object.

1. INTRODUCTION

1.1. Motivation

Berkeley UNIGRAFIX[1] started as a polyhedral modeling and rendering system[2] that could handle only objects bounded by planar faces — although of arbitrary complexity, with multiple contours and holes. However, most users of graphics systems soon tire of polygonal or polyhedral shapes and want at least rounded corners or, better yet, smooth looking continuous objects. Furthermore, CAD/CAM systems need to provide rounded surfaces since most mechanical parts are not entirely polyhedral; they contain circular holes, rounded edges, and curved fillets where two parts join.

In the creation of such objects, the user wants a simple-to-use interface that gives good results with an obvious way of specifying the round parts of the model. To stay within the framework of the Berkeley UNIGRAFIX system, we have looked at various possibilities for extending its boundary representation to curved objects, rather than taking a CSG approach with intrinsically round primitives. We have concentrated our attention on interpolating splines since they fit directly into the UNIGRAFIX paradigm where objects are described by vertices positioned in space and by edges and surface patches stretched between these points. In the modeling of technical parts, interpolating splines are also advantageous since often a few points, or even a whole profile, of the surface to be constructed are known precisely.

Local control over the surface shape is another desirable property. Often only a small part of a surface needs to be changed; one would then like the bulk of it to remain unaffected. We have tried to find interpolating splines that exhibit strong locality and integrate well with objects that are partially composed of flat surfaces. We want to avoid the overhead of representing all flat parts of an object also as spline surfaces.

These ideas are being implemented in a new modeling and rendering system called UNICUBIX. Its aim is to provide an environment for the non-mathematical user for producing geometrical objects of arbitrary shape and topology.

1.2. "Pleasing" Splines

Much work has been done in the area of interpolating splines and in the area of fitting curves and surfaces through a given set of points. The reader is referred to appropriate textbooks.[3-5]

Two pieces of work exemplify our own goals and have been particularly inspiring to the work described in this paper. Vaughan Pratt's work on conics for font definition[6] demonstrates how much can be achieved with using even very simple primitives. By a judicious choice in the selection of control elements, the individual spline patches can be made to join together in a very smooth and pleasing looking manner. Since our own goal is to provide smooth curves and surfaces in R^3 space, our lowest level primitives are cubic polynomials, rather than conics.

John Hobby's work on "Smooth, Easy to Compute Interpolating Splines"[7] demonstrates that very good results can be obtained by deviating from the strict formulation of natural splines. His curves, satisfying a mock-curvature constraint, give much more rounded and pleasing looking curves.

In the same spirit, we plan to achieve pleasing results with a minimum of

computational and conceptual overhead. We developed an approach primarily based on geometrical, rather than algebraic reasoning. The spline curve and surface patches are derived from a sequence of procedural steps that may include rules that take the special situation of a particular patch into account. This is more flexible than an approach that applies a fixed arithmetic expression to a given set of control points.

Our current implementation uses cubic Bézier curves for spline curve interpolation and for the boundaries between patches. It uses quartic triangular or bicubic quadrilateral patches that join with tangent-plane continuity to provide a smooth interpolating surface.

1.3. Paper Overview

In the first part of the paper (Sections 2 - 4) we will introduce the concept of our procedural step-by-step approach with the example of constructing planar interpolating curves through a given set of points. In Section 2 we first review the basics of spline segments. Then we present our procedural approach for the case of planar curves composed from cubic spline segments (Section 3). This procedure has two phases: first at each vertex the tangent direction of the curve is chosen; then the additional two Bézier points are chosen on these tangent lines. In Section 4 we give an overview how this approach can be extended to cases that go beyond the requirements arising in the construction of smooth shapes in UNICUBIX.

The second part of the paper (Sections 5 - 9) deals with surfaces in R^3 and starts with an overview of the approach (Section 5) followed by a review of the patches to be used (Section 6). In Section 7 we discuss the constraints that guarantee G^1-continuity.[8,9] For surfaces, our procedural approach has several phases (Section 8). First, the surface normals at all vertices are chosen. Second, the shape of the curved edge segments between the given vertices are determined. Third, the behavior of the tangent plane or the surface normal along the seams is defined. Then patches are fit into that network of curved boundaries by choosing interior control points that match the desired behavior on the seams and ensure G^1-continuity with the neighboring patches. In Section 9 we touch upon some rendering issues and review possibilities for creating Bézier rather than Gregory patches.

For more details about the UNICUBIX system and its implementation or about the underlying modeling issues the reader may request our technical reports on the subject[10,11] and should watch for forthcoming papers.

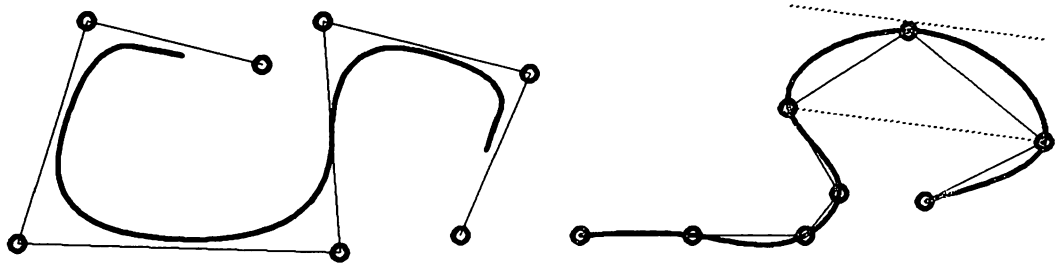
2. SHORT REVIEW OF SPLINES

This section briefly reviews a few concepts that are crucial for the understanding of our approach. More information can be found in introductory texts on geometric modeling, e.g.[4,5]

2.1. Approximating and Interpolating Splines

It is normally useful to distinguish between splines that interpolate the given vertices, and splines that only approximate these controlling points. A classical representative for an approximating curve is the B-spline. The coordinates of its control points are combined in a polynomial expression to define parametrically the coordinates of a point on

the curve as a function of the given parameter value. For the example of the cubic B-spline shown in Fig. 1a, each curve point is influenced by only four control points. This provides locality of control: if any control point is changed, at most four sections of the curve will be affected. If coinciding control points are avoided, the resulting curves are smooth. The cubic B-spline is twice differentiable and is thus said to be C^2-continuous. C^2-continuity automatically implies G^2, as well as C^1 and G^1-continuity. Another useful property is that the curve lies entirely within the convex hull of its control points.



(a) (b)

Figure 1: *Approximating (a) and interpolating (b) splines*

A different polynomial formulation can lead to an interpolating curve. Fig. 1b shows the example of a Catmull-Rom spline.[12,13] Again C^2-differentiability can be guaranteed with a suitable choice of weighting coefficients, but obviously the convex hull property holds no longer. While the B-splines can easily be made to have a very smooth appearance, the interpolating splines often show unnecessary wiggles and exaggerated asymmetric bulges, particularly when the control polygon has acute angles and sides of widely varying sizes.

In both cases it is often difficult for the user who places the control vertices to predict what the resulting curve is going to look like. In the case of B-splines, one does not know through what points the curve will go; and for the interpolating splines it is often difficult to guess the direction at which the curve passes through a given vertex. For the latter problem, some auxiliary construction may help. In the default formulation the tangent direction at an interpolated vertex can be found from the direction of the chord connecting the two nearest neighbors (see dotted line in Fig. 1b).

2.2. Bézier Segments

An alternative way of obtaining a smooth curve through a given set of points is to join individual curve segments together. The example shown in Fig. 2 uses three cubic Bézier segments. These are polynomial curves segments of degree 3 defined by 4 control points each. The first and last point give the start and end of the curve segment. Furthermore, the first two points together define the starting "velocity" and thus the tangent at the starting point. The last two points similarly give the ending velocity and tangent.

To join these segments together with tangent (G^1-) continuity,[8,9] requires that the last control point of one segment be shared with the first control point of the subsequent segment. Furthermore, the ending velocity vector of the first segment must be collinear

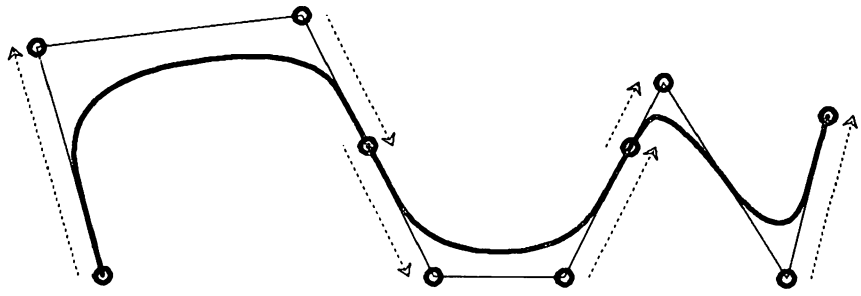


Figure 2: *Chain of Bézier-Segments*

with the starting velocity vector of the second segment. If, in addition, the velocities are equal in magnitude, the composite curve has C^1-continuity. But to give a smooth appearance, G^1-continuity is often sufficient.

Thus the task of producing a smooth curve is to find suitable control points for all segments in accordance with the above constraints.

3. RULE-BASED, GEOMETRICAL APPROACH TO SMOOTH CURVES

As an introduction to our multi-phase approach to the generation of smooth objects, we start with a procedure to obtain a smooth, pleasing looking curve through a sequence of vertices. In our demonstration case, the curve is composed from a sequence of cubic Bézier segments.

In a first phase, tangent directions are chosen at each vertex based on local information; in the case presented here, only the nearest neighbor vertices are taken into account. The second step is to place the inner Bézier control points on these tangent lines at a suitable distance from the given vertex. This determines the "velocities" of the parameterized point describing the curve on either side of the vertex and implicitly controls the bulge of the curve. G^1-continuity is ensured as long as the Bézier points on either side of the vertex lie on the same tangent line; for C^1-continuity they also would have to lie at equal distances from the vertex.

These selections are not simply made with a fixed algebraic expression involving the coordinates of the vertices, but may be based on several different computational procedures. The appropriate procedures are picked according to a set of rules that depend, for instance, on the constellation of the neighboring vertices. In particular, the end-vertices of a curve, or a cluster of collinear control points would be treated differently from other interior vertices.

3.1. Default Constructions for Tangent Direction

As shown in Fig 1b, Catmull-Rom splines derive the tangent direction of the curve at one of the interpolated vertices from the direction of the chord of the two nearest neighbor vertices. This may lead to a lopsided bulge if the two sides of the control polygon joined at this vertex are very asymmetric (Fig. 3a).

Another possible construction that avoids this problem simply defines the tangent

direction as being perpendicular to the angle bisector at this vertex (Fig. 3b). This method has the disadvantage that the tangent direction is completely insensitive to the lengths of the sides of the control polygon. Thus if the control polygon is a rectangle of any aspect ratio, the tangents will always pass through the corners at an angle of 45 degrees.

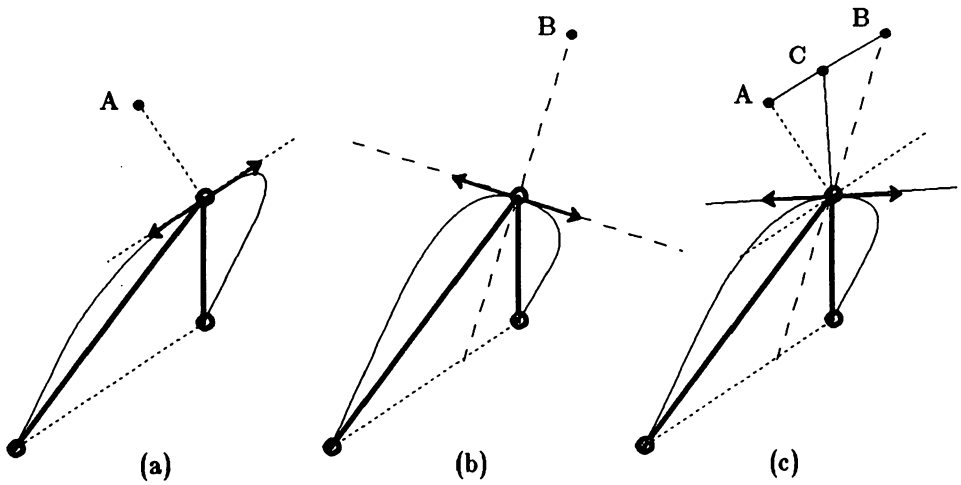


Figure 3: *Three ways of finding the tangent direction*

A good compromise can be reached by choosing the arithmetic average of the two tangent directions determined by the two methods above (Fig. 4c). The actual construction takes a normal vector of a length equal to half the chord to produce point A. It generates point B by reflecting the intersection point of the angle bisector with the chord about the central control vertex. Point C then represents the averaged "vertex normal"; the tangent direction is perpendicular to it. The weighting of the extreme solutions (a), (b) can occur in any desirable ratio. This ratio can affect the behavior of the curve locally or in a global manner, depending on whether the weighting ratio is set for each vertex individually or globally.

3.2. Default Selection for Velocity Magnitude

Once the tangent direction at each vertex has been determined, the next step is to select the velocities on either side of each vertex. This is equivalent to placing the inner two Bézier control points somewhere on the established tangent lines. For G^1-continuity it is not necessary that the velocities on either side of the vertex be the same. They can thus be influenced individually by the respective distances of the nearest neighbors. Again the most pleasing results over a wide range of constellations is obtained as a compromise between two extreme approaches (Fig. 4).

In Figure 4a the velocity is simply made proportional to the length of the underlying side of the control polygon. In Figure 4b both velocities on either side of the vertex are determined solely by the length of the chord between the nearest neighbors. Finally Figure 4c uses the geometric mean of the velocities determined in the above manner. In our view it gives very pleasing results in practically all cases examined.

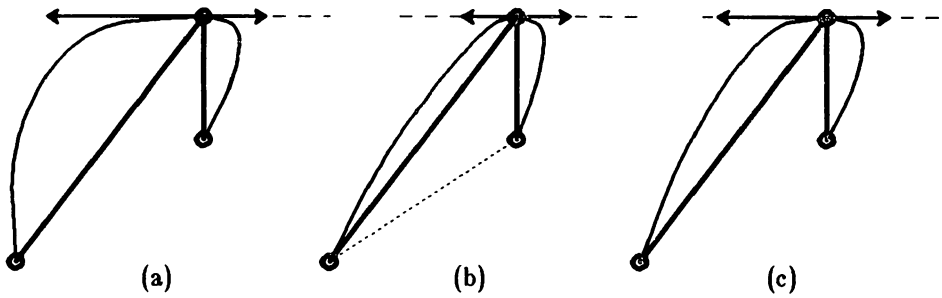


Figure 4: *Three choices for the velocity magnitude*

3.3. Shape Parameters

The procedures discussed above contain parameters that have been set to particular default values that seem to give suitable results in most cases examined. One such parameter is the weighting of the two extreme solutions for the choice of the averaged vertex normal direction (Fig. 3c). The value of this weighting factor will affect the general behavior of the tangents at the interpolated vertices. Another parameter is a multiplying factor on the calculated velocities. A bigger factor will produce more bulge of the curve segment between two vertices, while a smaller value will lead to a shorter segment that connects the two vertices more directly.

If these values are varied globally, the overall character of the whole curve can readily be affected in a uniform manner. Thus the power of shape parameters, developed for the more traditional splines,[14] can also be used in our procedural approach.

4. TOWARDS A RICHER RULE SET

Two simple procedures have been presented above to find the tangent directions and velocities at each vertex. The given rules are not sufficient to handle all constellations of vertices appearing in an unconstrained design in a satisfactory manner. To obtain "pleasing" results in all cases, the set of procedures and the set of rules to choose between them need to be enlarged.

4.1. Examples of More Sophisticated Rules

Special problems arise at the end-vertices of open curves and also when three or more control vertices are collinear, perhaps indicating that the user really wants a straight section through these points. Examples of rules that might deal with such situations are:

- If the control polygon does not have any inflections, then the interpolating spline should not have any inflections either.
- Three collinear vertices might indicate that the user really would like this piece to be straight; except when there are inflections, such as a sine wave defined by a triangular wave shape.
- When the control polygon has a very acute angle, the tangent of the curve should be almost perpendicular to the angle bisector.

- The vertex normal should never fall outside the extended lines of the control polygon at that vertex.

In the framework of the curved edges of UNICUBIX objects such problems normally do not arise. A more detailed discussion of these problems thus will be presented at some other time.[15] However, problems of a similar nature do occur in the decision steps for the creation of smooth surfaces in UNICUBIX (Section 8).

4.2. Measure of Quality, Desired Properties

The introduction of such "arbitrary" rules naturally raises the question: how should the results be evaluated?

The main goal is to obtain a "pleasing" behavior; therefore the users of the system will be the ultimate judges. We have made various attempts to obtain information from potential users. At the USENIX workshop the audience was asked to draw examples of "pleasing" interpolating splines through a given set of test vertices. One of the test shapes, interpolated with a Catmull-Rom spline (a) and with the simple compromise methods discussed above (b) is shown in Fig. 5.

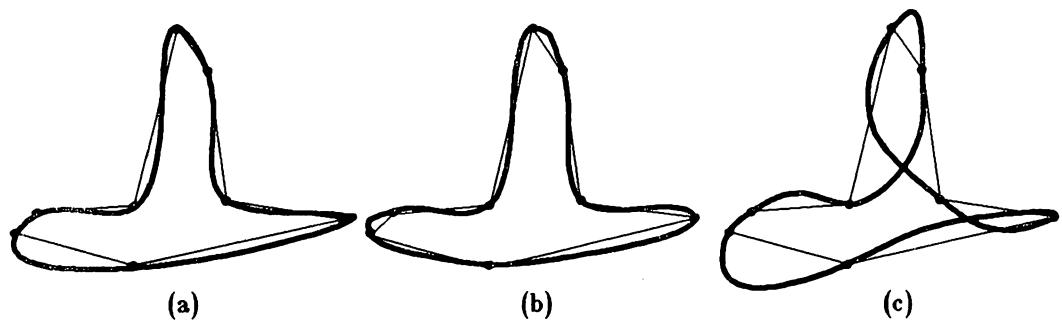


Figure 5: *Test pattern to find pleasing interpolating splines.*
A traditional spline (a), our procedural solution (b) and an artistic solution (c)

The results from the audience can typically be split in two categories: the "well-behaved" and the "artistic". The well-behaved solutions fall into a narrow range and are in good agreement with our solution displayed above (Fig. 5b). The other group displays exaggerated "loopy" curves (Fig. 5c) — often not even passing through the vertices in the sequence indicated by the control polygon — and they vary too strongly to indicate any particular trend. There is strong suspicion that these "artists" were more motivated to be original than to seriously try to find some form of minimal spline through the given vertices.

Obviously, the definition of the "best" or "most suitable" spline is application dependent. A smooth surface to be fabricated on a milling machine may be different from a surface whose primary objective is to perform well in a wind tunnel. One may have to define several application-dependent rule sets or must make the rule sets modifiable by the user. Furthermore, any practical interactive design system must allow the user to move the individual control points in areas where the default solution curve is insufficient.

5. OVERVIEW OVER SURFACE INTERPOLATION IN 3D

An equivalent procedural approach can be taken to make pleasing surfaces that interpolate a given set of points in 3D. This is an outline of the procedural steps:

1. Define the surface by placing vertices in R^3 and by using edges to indicate how the latter should be topologically linked by the surface.
2. Select the *vertex normals* - the normals of the surface at the vertices.
3. Place the Bézier points in the tangent planes at the vertices. The goal is to replace the original edges with suitable cubic curves that will lie in the surface to be constructed.
4. Define the surface normal or the tangent plane along the chosen cubic curves or - equivalently - determine the behavior of the surface seams along the borders between the various patches.
5. Fit into each mesh surrounded by cubic boundaries a Gregory patch that blends with its neighbors with G^1-continuity.
6. If desirable, subdivide the Gregory patches into Bezier patches.

The techniques to join patches with G^1-continuity have been strongly influenced by the work by Farin[16,17] and by Chiyokura.[18,19] Each step in the above sequence ensures that the constraints applicable up to this point are met, so that subsequent steps do not run into inconsistencies. Again, variations in the procedures can produce objects of varying overall character.

6. BEZIER AND GREGORY PATCHES

As pointed out earlier, we use quadrilateral as well as triangular patches. This allows us to handle nets of arbitrary topology and to find a surface description that easily matches the natural symmetries of the intended shape. In this section we briefly review these patches.

6.1. Quadrilateral Patches

One of the basic surface elements used in UNICUBIX is a quadrilateral bicubic vector-valued tensor-product Bézier patch. Such a patch is defined with sixteen control points, 4 in the corner vertices, 2 along every edge, and 4 interior points (Fig. 6).

All but the 4 interior points are already given by the time we try to fit a patch into the mesh of cubic boundaries. It turns out that in the general case the four interior points do not offer enough freedom to provide G^1-continuity across all four borders. Instead we use a bicubic Gregory patch [20] which provides twice as many interior control points (Fig. 7). Every interior point is split into a pair of points where one each controls the behavior of the patch on one of the two borders coming together in the associated corner. The influence of these two points is linearly interpolated as surface points gain more distance from one border and approach the other. For instance, in the corner where the borders $u=0$ and $v=0$ come together, the weighting of the the Gregory point pair is given simply by the ratio of the parameter values of u and v respectively. Because of these split control points, the Gregory patch does not have the twist constraints that one finds in the Bézier patches. On the other hand, it is no longer a purely polynomial expression in the parameters u and v.

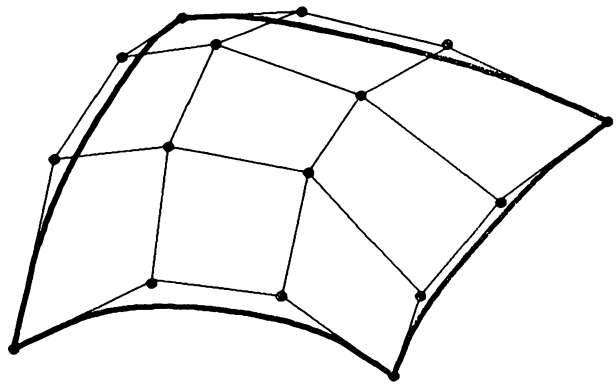


Figure 6: *Bicubic Bézier patch with its 16 control points*

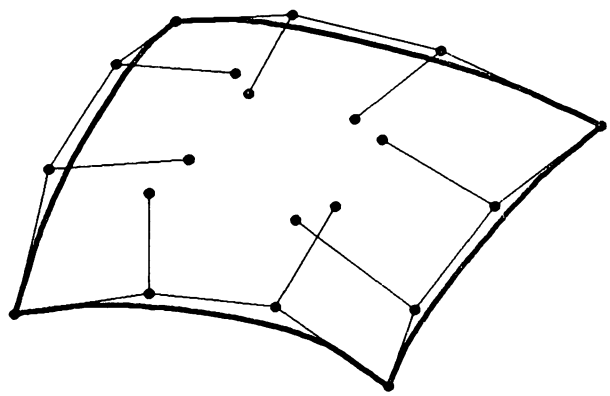


Figure 7: *Bicubic Gregory patch with its 20 control points*

6.2. Triangular Patches

Triangular patches are best described in barycentric coordinates. In this representation a point in the plane of the triangle is expressed as a linear combination of the three vertex coordinates; the three coefficients always sum to 1.

A picture of a cubic patch is shown in Fig. 8a. Such a patch has the same number of control points on the edges as the bicubic patch discussed above, but it has only a single interior control point. To provide two interior control points along each edge as in the above case, we have to use a quartic triangular patch which provides a total of three interior control points (Fig. 8b).

Furthermore, to permit us to set two interior control points independently from the constraints on each boundary, we need to split these control points as in the quadrilateral case. This leads to a triangular version of the Gregory patch (Fig. 9). Again, in the expressions that describe the surface in parametric form, the pairs of control points enter as a linearly interpolated expression that is weighted with the ratio of the two relevant parameters of the barycentric coordinate space.

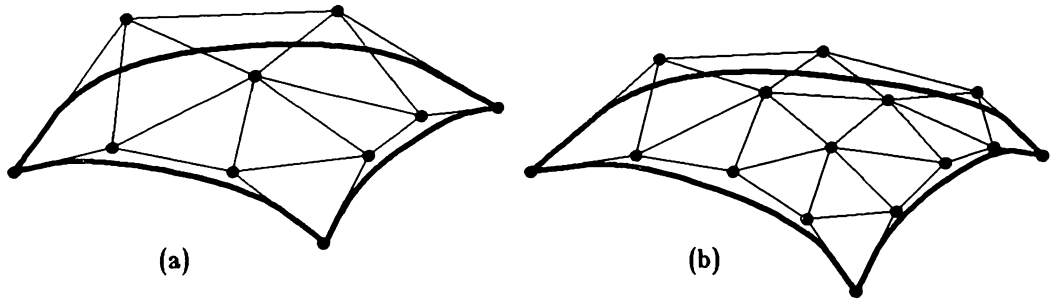


Figure 8: *Cubic (a) and quartic (b) triangular Bernstein patches.*

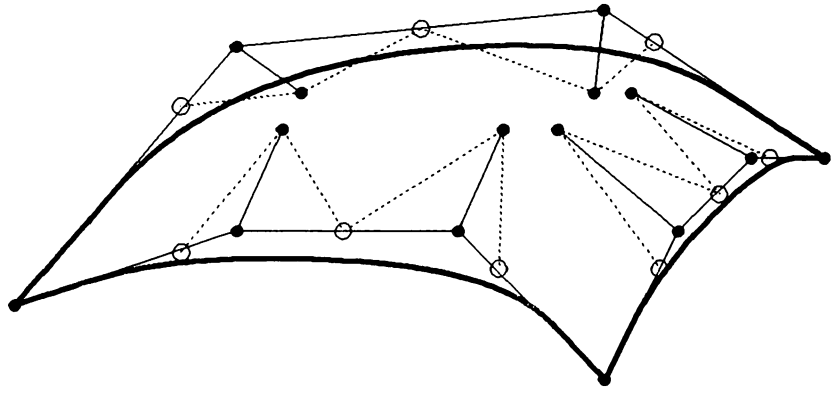


Figure 9: *Triangular quartic Gregory patch with its 18 control points*

Since the quartic Bernstein or Gregory patches have quartic borders with one more control vertex along their sides than the given cubic borders, we need to *degree elevate* our cubic borders to degree four. This is a simple process. The three new control points (○) can be found as simple linear expressions of two of the old control points (•) (Fig. 10). Intuitively they can be viewed as lying at "equidistant" intervals on the control polygon — if the sides of the control polygon are assumed to be of equal length.

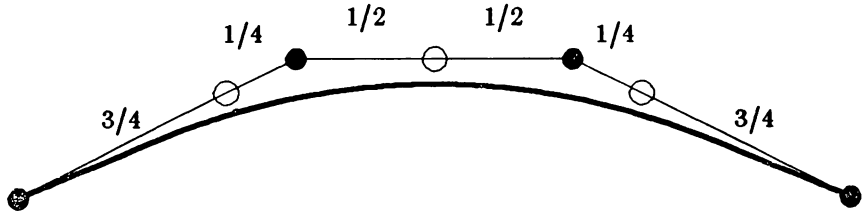


Figure 10: *Degree elevation from a cubic* (•) *to a quartic* (○) *control polygon*

With these descriptions, triangular as well as quadrilateral patches can be spliced readily to the given cubic borders (Fig. 11). As we will see in the next section, when we

determine the two interior patch control points along a boundary, we don't even have to know whether this is for a quartic triangle or for a bicubic quadrilateral patch.

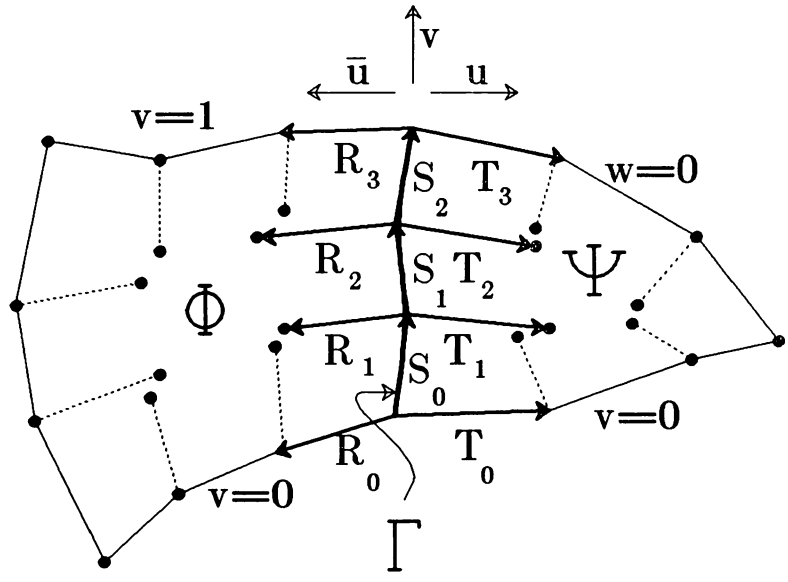


Figure 11: *Joining a quadrilateral and a triangular patch*

7. JOINING PATCHES WITH TANGENT-PLANE CONTINUITY

This section reviews several mathematical formulations for tangent-plane (G^1) continuity between adjacent patches. We explore the constraints and the number of degrees of freedom in the placement of the interior control points of the patches.

7.1. Expressing Continuity

First-degree geometric continuity (G^1-continuity) requires that both patches Φ and Ψ (Fig. 11), meeting along the boundary Γ have the same tangent plane at every point of the common boundary curve ($v \in [0,1]$). Thus for any given parametrization of the patches, the two cross-boundary derivatives $D\Phi$ and $D\Psi$ and the common derivative along the boundary $D\Gamma$ — the tangent of the boundary curve — need to lie in one plane, but the two cross-boundary derivatives need not be collinear. One elegant way to express G^1-continuity is to set the determinant of these three derivatives to zero:

$$\det (D\Phi(v), D\Psi(v), D\Gamma(v)) = 0, \quad v \in [0,1]. \tag{1}$$

This requires the selection of suitable cross-boundary derivatives. Patches Φ and Ψ can be either bicubic quadrilateral or quartic triangular patches. For the quadrilateral case the choice is obvious: one uses the partial derivative with respect to the parameter that is kept constant on the boundary curve:

$$D\Phi(v) \equiv \Phi_u(\bar{u}=0,v). \tag{2}$$

For a barycentric triangle the situation is more complicated. We follow Farin[16] in

defining a *radial* derivative:

$$D\Psi(v) \equiv (1 - v)\,(\Psi_u - \Psi_v) + v\,(\Psi_w - \Psi_v). \tag{3}$$

A graphical interpretation is that the cross boundary control vectors are simply given by connecting the Bézier points of the cubic border with the corresponding interior Gregory points of the quartic triangle (see also Fig 9).

In another formulation we can express the coplanarity of the three derivatives as a vanishing vector function:

$$\alpha(v)\,D\Phi(v) + \mu(v)\,D\Psi(v) + \lambda(v)\,D\Gamma(v) = 0. \tag{4}$$

Since $D\Phi(v)$ and $D\Psi(v)$ are of degree 3, while $D\Gamma(v)$ is of degree 2,

$$\mathbf{deg}\,(\alpha(v)) = \mathbf{deg}\,(\mu(v)) = \mathbf{deg}\,(\lambda(v)) - 1. \tag{5}$$

This is now a system of vector equations which decomposes into three identical scalar systems, one for each coordinate. This is simpler than the determinant equation above which mixes the components along different axes. If we further restrict the degree of the polynomial functions $\alpha(v)$ and $\mu(v)$ to degree two, the equation system becomes quite manageable.

Either of the above two formulations ties the control points on one side of the curve to the points on the other side. Patches on either side can thus not be chosen independently. In the next section we explore how many degrees of freedom we have in this system from which we want to obtain the twelve coordinate values for the two pairs of control points on either side of the border.

7.2. Degrees of Freedom

At this point, it is useful to analyze the situation on the border curve a little more closely. We would like to know how many degrees of freedom there are intrinsically before we start using them for the sake of obtaining simpler solutions. Ideally we would like to keep them all and express them in terms of natural shape parameters that could be set globally or locally and through which the user can control the shape of the surface in an intuitively obvious manner. This formulation is the subject of ongoing research.[21]

The above determinant equation (eqn 1) is a polynomial of degree 8 in v, which yields 9 equations for the coefficients of the various degrees of v. Two of these equations represent the endpoint conditions that all curves at a vertex must end in a common tangent plane. These will be automatically fulfilled if the two middle Bézier points of the boundary curves have been properly selected. Thus there remain 7 equations for the 4 vectors $\mathbf{R}_1$, $\mathbf{R}_2$, $\mathbf{T}_1$, $\mathbf{T}_2$ (12 unknown scalars). They proved to be so complicated, that we could not cast them into a usable form[10] even with the use of VAXIMA.[22] However, the system of equations shows that there are only five degrees of freedom in placing the 4 control vertices. The same result follows from the vector equation (eqn 4).

This means that we cannot select one patch at will and then fit the other one to it with G^1-continuity. The exact nature of the restriction is not currently understood. However certain construction methods will lead to a solvable system of equations. In particular, certain implicit or explicit assumptions about the behavior of the seam along the shared border will lead to equation systems that are not hard to solve.

7.3. Specifying the Behavior of the Seam

We can use some of the available degrees of freedom to nail down the behavior of the surface at the border and then solve for the position of the interior control points in accordance with the chosen behavior.

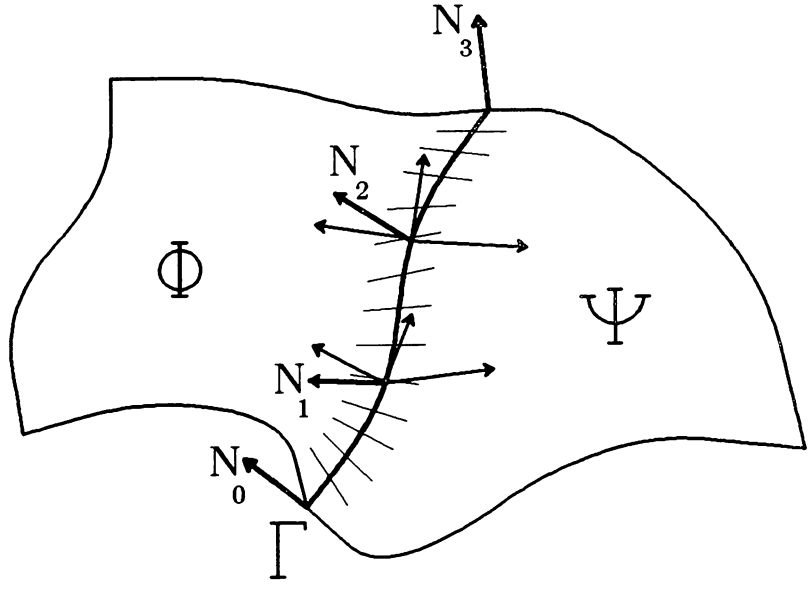


Figure 12: *Normals and tangents on the seam between two patches*

In one approach, the behavior of the *surface normal* $\mathbf{N}(v)$ is specified along the border, and the patches on either side then have to be constructed so that their tangent planes are perpendicular to the given normal direction at every point. This constraint is best expressed as a vanishing dot-product between the normal and the cross-boundary derivative at any point along the border $\mathbf{\Gamma}$:

$$(\mathbf{N}(v), D\mathbf{\Phi}(v)) = 0, \quad v \in [0,1]. \tag{6}$$

Of course, if we prescribe the normal along the border, it must be done in such a manner that at every point it is perpendicular to the curve's tangent:

$$(\mathbf{N}(v), D\mathbf{\Gamma}(v)) = 0, \quad v \in [0,1]. \tag{7}$$

This looks like the most promising approach to extract some explicit shape parameters. We are currently studying the various possibilities of defining a default behavior for the normal vector.

In another approach, a generic *cross-boundary tangent* $\mathbf{T}(v)$ is defined, that together with the border tangent $D\mathbf{\Gamma}(v)$ defines the tangent plane at every point. The two patches then have to match this tangent behavior, or — equivalently — their cross-boundary derivatives must be linear combinations of the border tangent and the defined cross-boundary tangent:

$$\alpha(v)\, D\mathbf{\Phi}(v) = \lambda(v)\, D\mathbf{\Gamma}(v) + \mathbf{T}(v), \tag{8}$$

$$\mu(v)\, D\mathbf{\Psi}(v) = \rho(v)\, D\mathbf{\Gamma}(v) + \mathbf{T}(v). \tag{9}$$

In order to obtain a generic tangent behavior, Chiyokura starts by setting the cross-boundary tangents at the two endpoints to $\mathbf{T}_0 - \mathbf{R}_0$ and $\mathbf{T}_3 - \mathbf{R}_3$, respectively (see Fig. 11). Although the cross-boundary tangent can be cubic in general, Chiyokura linearly interpolates it between the two values at the ends of the boundary curve. With this approach the degree of the polynomial equations from the vanishing vector function (eqn 4) is reduced drastically, and the computation for the two interior control points of a patch becomes quite simple. This procedure is performed separately for the two patches on either side of the border curve.[18,19]

Our current implementation of UNICUBIX follows this last approach. Other approaches are being studied and their trade-offs are evaluated. We are also investigating how to extract the most useful geometric shape parameters from this approach.

8. CONSTRUCTION RULES IN UNICUBIX

UNICUBIX has a mode in which it automatically converts a polyhedral object into a smooth surface through the given vertices. The process follows the step-by-step procedures outlined in Section 5.

8.1. Choice of Vertex Normals

The first task is to choose suitable vertex normals (Fig. 13). Obviously these normals should reflect any inherent symmetries of the polyhedral shape. Furthermore, they should be tessellation independent. Since our program can only handle triangular and quadrilateral patches, polygonal faces with more sides must be subdivided first. This is done by an automatic tessellation program that returns triangles or convex quadrilaterals. Since this tessellation is performed with a sweep plane algorithm, the result, and, in particular, the number of cut-lines coming together at one vertex, is dependent on the orientation of the object in space. However, we would like the shape of the final smooth surface to be orientation independent.

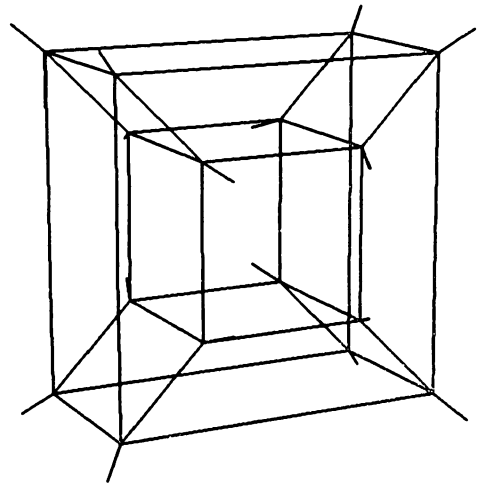


Figure 13: *Polyhedral frame of square torus with vertex normals*

A procedure that achieves this goal is to average all face normals of the polygons that share a vertex, weighing the individual contributions with the angle of the polygon at that vertex. Any extra cut in a face will simply produce two smaller contributions summing to the same result.

8.2. Defining the Cubic Borders

The next step is to choose two inner Bézier points for each curve segment replacing the original straight edges. These points must lie in the tangent plane of the topologically nearest vertex; within that plane each vertex has two degree of freedom.

In our first implementation we have taken a very simple approach. Each edge leaving a particular vertex is projected onto its tangent plane. The corresponding Bézier point is then chosen at a distance from the vertex equal to one third the length of the original edge. This gives good results for convex bodies but is not quite satisfactory for the inner parts of a torus or similar concave regions. More elaborate rules to deal with this case are under development.

Once the Bézier points have been determined, the cubic border curves can be drawn; they are displayed in Figure 14 together with the vertex normals.

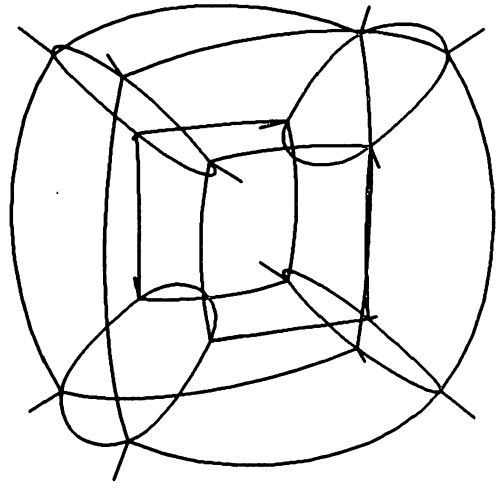


Figure 14: *Cubic boundary curves generated from the frame of Figure 13*

8.3. Specifying the Seams

In our first implementation we have followed Chiyokura's approach,[19] specifying in an implicit manner the behavior of the surface patches along their seams (see Section 7.3). Cross-boundary derivatives are derived from the Bézier points associated with the end-vertices of the cubic border curves and linearly interpolated along these curves. Alternatives involving the surface normals are under investigation.

Of course, not all borders need to be smoothly interpolated by the adjacent patches. Objects in UNICUBIX can display a mixture of rounded and of sharp edges. For the case of sharp edges, the procedures above need to be modified suitably. In particular, the determination of the vertex normals changes. At any vertex there can now be more than

one distinct normal vector, associated with different faces coming together with different tangent planes.

The procedure to determine the control points for the cubic edges must then pick the proper normal carefully. But afterwards, the determination of the interior patch control points proceeds along the same scheme unless the patch happens to be a completely flat face. In this case, it will be treated as a many-sided planar polygon.

8.4. Filling in the Patches

The specification of the seams implicitly defines the tangent behavior of the patches at all their boundaries. By using Chiyokura's formulas,[19] we can generate the two interior Gregory points along every border curve (Fig. 15). This then defines the patch fully.

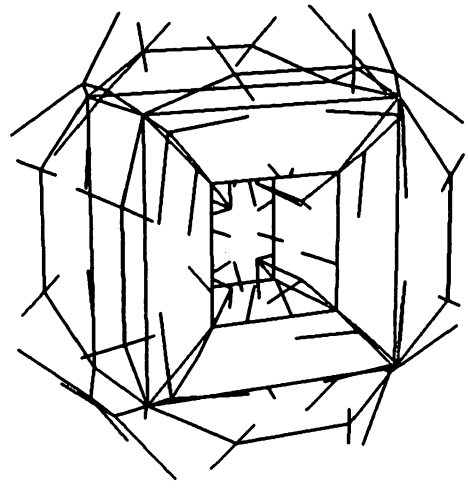


Figure 15: *Cubic border frame with all control points*

We can now evaluate the patch at a regular grid of points in the rectilinear or barycentric triangular parameter space, and connect these points on the surface with a mesh of straight edges and triangular or quadrilateral faces. The result is a net-representation of the surface patches as shown in Figure 16.

9. RENDERING ISSUES

9.1. Results

The net representation in the previous section is a regular UNIGRAFIX description [1] of a polyhedral object — provided that the facets are planar. Triangles, of course, are no problem, but with quadrilaterals we cannot be sure. Often the inherent symmetry of the object assures that the quadrilaterals are also planar; these are the preferred cases for quadrilateral patches. In other cases one may use a net fine enough so that the deviation from planarity causes no problem in the renderer (Fig 17). One of our renderers, ugdisp,[23] has a special option to accommodate somewhat warped polygons. This same renderer also has the capability to do smooth Gouraud shading of polyhedral objects.

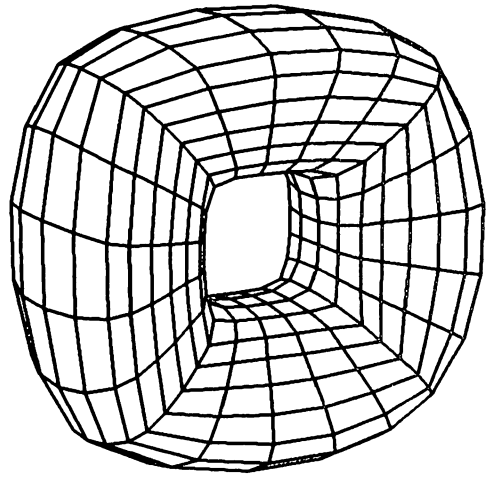


Figure 16: *Net-representation generated from Fig. 15*

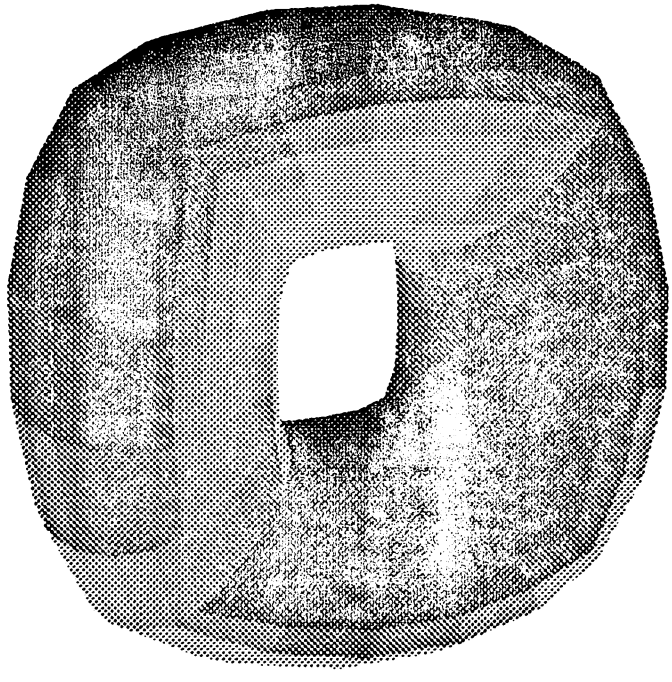


Figure 17: *Shaded rendering of object shown in Figure 16.*

9.2. Conversion to Bézier Patches

Bézier patches are more attractive than Gregory patches for a variety of reasons. Their polynomial form leads to computational efficiency in rendering. Furthermore, there is a wealth of published and implemented algorithms for subdivision, rendering, and mutual intersection of such surfaces, which are not yet available for the case of Gregory patches. We have thus tried to fill the meshes between our cubic boundaries with Bézier

patches. However, this cannot be achieved with single Bézier patches of any degree because of the twist constraints at the corners. The boundary curves would have to be subjected to extra constraints to make possible the use of Bézier patches in each mesh.

By using Gregory patches we have obtained complete locality of control. The cubic boundaries can be specified with few constraints, and the patches can then be fitted into the meshes individually. Ideally we would like to retain this property. To do so with Bézier patches, we must partition each mesh and introduce at least one new boundary at each corner of the mesh. The exact shape of these internal boundaries must be left under the control of the system so that the twist constraints of the Bézier patches can be fulfilled automatically.

Using this general approach, we have found a way to fill triangular meshes with three Bézier triangles rather than with one Gregory patch (Fig. 18a). Similarly we can replace a quadrilateral Gregory patch by either five quadrilateral Bézier patches (Fig. 18b), or by a mixture of triangles and quadrilaterals.[11]

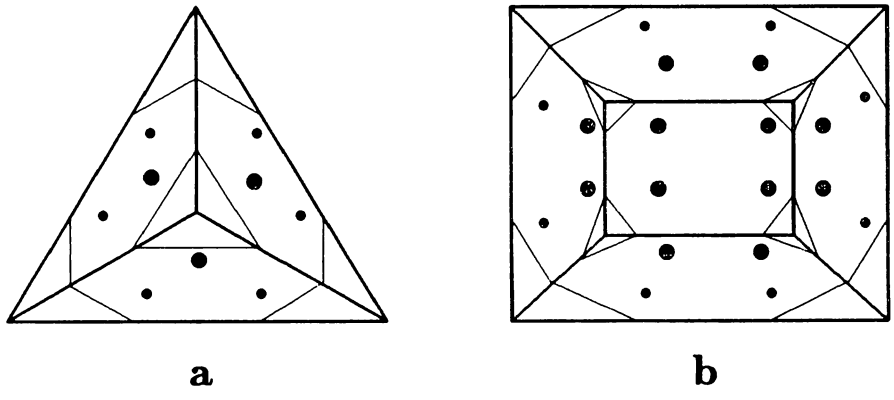


Figure 18: *Replacing Gregory patches with multiple Bézier patches*

In all cases, along every outer boundary of the original mesh two interior patch control points are determined with the same techniques as used to find the two Gregory points (Section 8.4); these are the small dots in Figure 18. The remaining interior control points (large dots) and all the control points on the internal boundaries are then determined so that one obtains a system of three to five Bézier patches that join with G^1-continuity and meet the specification of the external seams.[11]

The replacement of a quadrilateral with four Bézier triangles so far has resisted our attempts to find a solution.

10. CONCLUSION

We have implemented a prototype of a practical modeling system for the non-mathematical user. UNICUBIX has a very terse description of the objects since it needs to store only the information for the shape of the cubic boundary curves. All information for the shapes of the patches is implicitly given by the shape of the boundary curves and by some global (or possibly local) shape parameters.

This implicit information is extracted from the given UNICUBIX description in a sequence of procedural steps. In turn, the vertex normals are constructed, then the inner two Bézier points for the cubic borders are determined if they are not explicitly given, from that the interior patch control points can be computed, and finally a net representing the surface patch can be constructed.

This same procedural approach could be extended to curves and surface patches of higher order, making it possible, for instance, to obtain interpolating splines with curvature continuity.

11. ACKNOWLEDGEMENTS

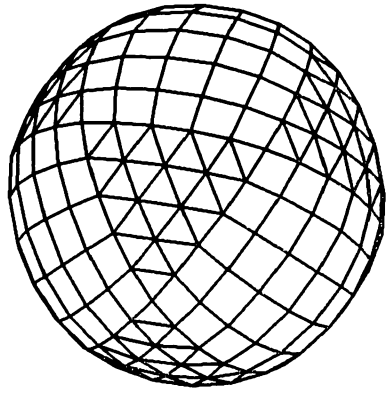
This development has been made possible by the dedicated effort of Lucia Longhi and Leon Shirman, who worked out the mathematical foundation for the described approach and who created the UNICUBIX programs. Thanks also go to Brian Barsky and Mark Segal who read this manuscript on very short notice. This effort is supported by Tektronix, Inc. and by the Semiconductor Research Corporation.

REFERENCES

1. C.H. Séquin, "Berkeley UNIGRAFIX, A Modular Rendering and Modeling System," *Proc. of the 2nd USENIX Computer Graphics Workshop*, Monterey CA, pp. 38-53 (Dec. 1985).
2. C.H. Séquin and P.S. Strauss, "UNIGRAFIX," *Proc. 20th Design Automation Conf.*, Miami Beach, FL, pp. 374-381 (June 1983).
3. I. D. Faux and M. J. Pratt, *Computational Geometry for Design and Manufacture,* Ellis Horwood Ltd (1979).
4. M. E. Mortenson, *Geometric Modeling,* Wiley, New York (1985).
5. R. H. Bartels, J. C. Beatty, and B. A. Barsky, *An Introduction to the Use of Splines in Computer Graphics,* Morgan-Kaufmann Publishers, Inc., Los Altos, California. To appear.
6. V. Pratt, "Techniques for Conic Splines," *SIGGRAPH'85 Conf. Proceedings*, pp. 151-159 (July 1985).
7. J. Hobby, "Smooth Easy to Compute Interpolating Splines," *Discrete and Computational Geometry* **1**, pp. 123-140 (1986).
8. B. A. Barsky and T. D. DeRose, "Geometric Continuity of Parametric Curves," Tech. Report, U.C. Berkeley (Oct. 1984).
9. A. Fournier and B. A. Barsky, "Geometric Continuity with Interpolating Bézier Curves (Extended Summary)," pp. 337-341 in *Proceedings of Graphics Interface '85,* Montreal (27-31 May 1985). Revised version published in *Computer-Generated Images -- The State of the Art,* edited by Nadia Magnenat-Thalmann and Daniel Thalmann, Springer-Verlag, 1985, pp. 153-158.
10. L. Longhi, "Interpolating Patches Between Cubic Boundaries," Master's Report, U.C. Berkeley (Dec. 1985).
11. L. Shirman, "Symmetric Interpolation of Triangular and Quadrilateral Patches

between Cubic Boundaries," Tech. Report, U.C. Berkeley (Dec. 1986).

12. E. E. Catmull and R. J. Rom, "A Class of Local Interpolating Splines," pp. 317-326 in *Computer Aided Geometric Design*, ed. R. E. Barnhill and R. F. Riesenfeld, Academic Press, New York (1974).
13. T. D. DeRose and B. A. Barsky, "Geometric Continuity and Shape Parameters for Catmull-Rom Splines (Extended Abstract)," pp. 57-64 in *Proceedings of Graphics Interface '84*, Ottawa (June 1984).
14. B. A. Barsky and J. C. Beatty, "Local Control of Bias and Tension in Beta-splines," *ACM Transactions on Graphics* **2**(2), pp. 109-134 (April, 1983). Also published in *SIGGRAPH '83 Conference Proceedings* (Vol. 17, No. 3), ACM, Detroit, 25-29 July, 1983, pp. 193-218.
15. C. H. Séquin, "Rule Based Spline Interpolation," work in progress.
16. G. Farin, "A Construction for Visual C1 Continuity of Polynomial Surface Patches," *Computer Graphics and Image Processing* **20**, pp. 272-282 (1982).
17. G. Farin, "Triangular Bézier-Bernstein Patches," *Computer Aided Geometric Design* **3**(2), pp. 83-127 (1986).
18. H. Chiyokura and F. Kimura, "Design of Solids with Free-form Surfaces," *Computer Graphics (Siggraph'83 Conf. Proc.)* **17**(3), pp. 289-298 (1983).
19. H. Chiyokura, "Localized Surface Interpolation Method for Irregular Meshes," *Proc. Computer Graphics Conf.*, Tokyo (1986).
20. J. A. Gregory, "Smooth Interpolation Without Twist Constraints," pp. 71-87 in *Computer Aided Geometric Design*, ed. R. E. Barnhill and R. F. Riesenfeld, Academic Press, New York (1974).
21. C. H. Séquin, "Shape parameters for G1 continuous patches," work in progress.
22. R. J. Fateman, "Addendum to the MACSYMA Reference Manual for the VAX," Tech. Report, CS Div., U.C. Berkeley (1982).
23. N. Gal, "Hidden Feature Removal and Display of Intersecting Objects in UNIGRAFIX," Master's Report, U.C. Berkeley (Jan. 1986).



Porting Unix to the Bösendorfer

Michael Hawley
MIT Media Lab E15-493
20 Ames St, Cambridge MA 02139
media-lab.mit.edu!mike

Introduction

What would you do with a $70,000 1-ton 9-foot 6-inch Bösendorfer Imperial Concert Grand piano that is controlled by a high-quality computerized recording and playback device?

Obviously, the first thing would be to

```
int pianofd = open("/dev/bosendorfer", O_RDWR);
```

though it will sadly take time before a proper device driver is concocted —
there are a few technical hurdles to clear before musically interesting work can be acheived.
This talk discussed the piano, old-fashioned player pianos,
and what we hope will be done with an instrument of this potential.

The Piano

Several years ago, Kimball purchased the Bösendorfer piano company. After seeing a prototype computerized piano built by Wayne Stahnke, and with some impetus from Marvin Minsky and John Amuedo at MIT, in 1983 they commissioned Stahnke to build the highest-quality player-recorder piano possible.

A player piano is obviously limited by the quality of the instrument in which the mechanics reside. This is why a number of manufacturers of player pneumatics installed their products, OEM-style, in instruments of various makes. For instance, Ampico put their action in pianos made by Fischer, Mason and Hamlin, Chickering, and Grotrian-Steinveg, to name a few. Their highest high-end instrument was undoubtedly the Bösendorfer. If American Steinways were Cadillacs, and Bechsteins were top-of-the-line BMW's, Bösendorfers would be the Rolls Royce of pianos - meticulously crafted (much by hand), the instrument is large and truly opulent. Curiously, the piano has 97 keys (9 more than the usual 88, down to low C) and the bonus bass notes provide additional resonance. A stand-alone (that is, non-computerized) Bösendorfer generally retails for $65,000 to $80,000 depending on the Deutsch Mark; think of it as one and a half Sun-3's, or almost two American Steinway concert grands. Bösendorfer has been a favorite of composers and pianists - Liszt, Brahms, Bartok, Mahler, and Strauss all owned or preferred Bösendorfers.

Stahnke's player-recorder mechanism uses precision solenoids to drive the player (one for every note, three for the pedals) and a system of optical gates to record. Attached to every key are two opaque "flags", one under the key and one on the hammer shank. These interrupt optical gates, and from the difference one computes the velocity of the hammer as it strikes the string. In this way, the recording mechanism captures data from all keys and pedals. A Z-80 controller receives

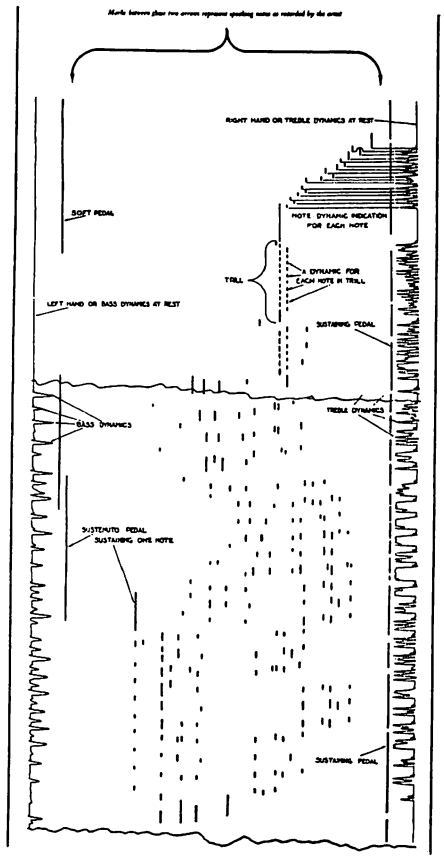
time-tagged events for each note (onset velocity, and release time, sampled at 800 Hz) and each pedal (a range for the soft and damper pedals, binary for the middle pedal, sampled at 100 Hz). The only important "missing" data, in a sense, is the key release velocity, but given the mechanics, this is physically not computable (since the hammer shank rebounds below the gate after it strikes a note) and musically not terribly significant, except for the most subtle kinds of note-releasing.

At the moment, a 25-pin connector cable comes out of the piano and into the Z-80, which sits in a multibus card cage from whence it has access to a 9" floppy disk controller and a CPM operating system. A few basic applications run on the CPM machine - play, record, a simple line-editor, etc - but there is currently no easy way to process data or control the piano using a nice computer (read, Unix workstation with graphics). In fact, owing to the baroque nature of 9" floppies, no one seems to have physically transported the data to any other machine. Although MIDI (musical instrument digital interface, akin to RS232+ASCII for synthesizers) instruments and controllers are more convenient, current controllers lack the dynamic range or temporal resolution of the Bösendorfer, and synthesizers don't sound as nice. (Yes, Kurzweils and other sampling synthesizers emulate pianos, but listening to a sampled piano with badly modeled resonance and pedalling does not compare to putting your head inside the Bösendorfer when it is running). Besides which, there is a certain unmistakeable mechanical charm associated with a real instrument, and real keys and hammers flying. Obviously, though, some kind of MIDI i/o port and a fast computer interface are desirable. A good (better than Z-80) controller would probably be some ilk of off the shelf 68000 i/o board with a pair of ports clocked to run at MIDI data rates (the Z-80 is just not fast enough to monitor the piano and talk to outside devices at the same time.)

Player Pianos in the Good Old Days

Player piano technology was born in 1908, boomed, and faded away just after the stock crash. (McLuhan would probably point out that the rise of media like radio, telephones, and later television absorbed that market and were a factor in the demise of this quaint medium). The very first player piano was really a *piano-player* - an 88-fingered box built by the Welte-Mignon company, it would fit on any piano. This kind of modularity wasn't very important for consumers, most of whom only needed one piano in their home, and besides, it made it impossible for the human to play duets with the machine, so the mechanism migrated inside the piano. The early players (and later inexpensive ones) were driven by *pneumatics* - a stack full of little vacuum-powered bellows. Pumping the pedals created a vacuum in the "stack", drove a pneumatic motor to move the roll, and whenever a note-hole crossed the tracker bar, triggered a valve opening a note-bellows to the vacuum in the stack, causing it to suck shut and play a note.

Reproducing player pianos - pianos capable of reproducing the subtle dynamic inuendos of performers, as opposed to constant-velocity "pianolas" - flourished in the 20's and reached quite a high mechanical art. The technology to do this with air and punched paper not much wider that 88 holes was tricky: for instance, most "stacks" were divided into treble and bass halves, the dynamics (vacuum pressure) of which could be independently controlled by an intricate bellows with 4 chambers and roughly 24 "bits" of dynamic range. Bringing out a melody line requires instantaneously varying the pressure to strengthen a note (the note onsets are usually slightly skewed, ie, melody notes don't usually coincide exactly with other harmonic tones); the following Welte recording shows some of this:



This photograph of a section from an original recording of Chopin's Etude in F Major shows how every detail of the artist's playing is graphically recorded while he plays. ¶With this absolutely authentic "tone picture" as a guide, the making of records for Welte Mignon (Licensee) Reproducing Pianos is free from every vestige of guesswork. Nothing is added or substracted that the artist does not himself put into his music, so that the record is not a mere approximation, but an exact reproduction of his playing. ¶What may be called the "film of the music camera" receives impressions of every detail of both his fingering and pedaling. ¶The exact position of every note played is fixed by faint vertical lines corresponding in number to the keys on the piano. The staggered lines on the extreme right and left are the means by which the mechanism, like the delicate needle of the seismograph that records the slightest tremor of the earth, graphically indicates exactly the degree of pressure with which the artist struck the keys, thus faithfully recording the finest shading of his interpretation.

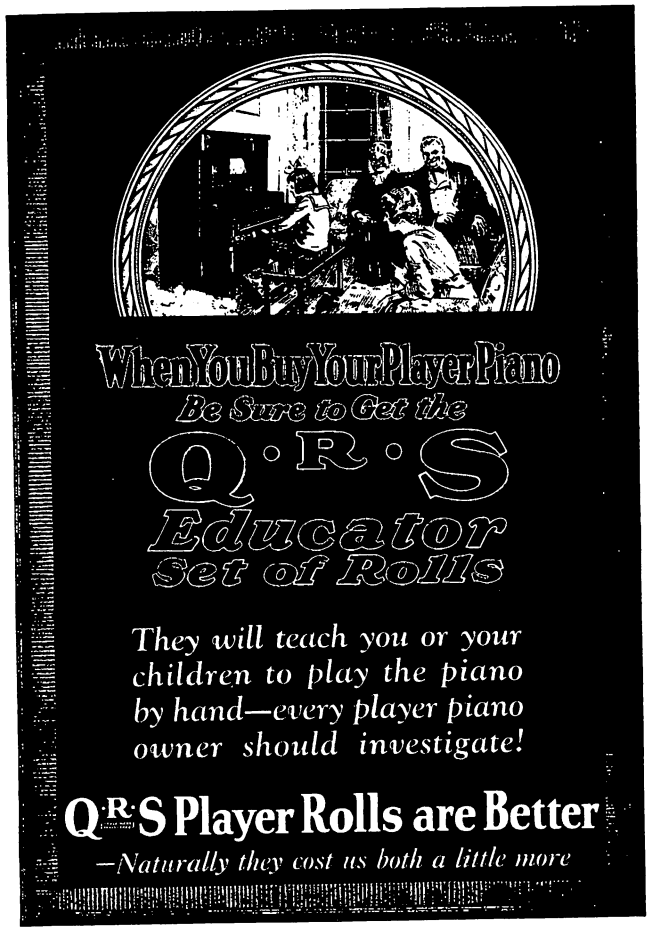
This sort of multiplexing (two pressure channels quickly controllable) worked remarkably well - in fact, it satisfied concert pianists. Publicity stunts ("is it live, or is it ... a Duo-Art reproducing grand") were popular, and serious promotional concerts were often given with player pianos as soloists. One review caught my eye, since the pianist, Leo Ornstein, was Severo Ornstein's father, and Severo (with John Maxwell) built the *Mockingbird* music scoring editor for the Dorado (a project which should be well-known to interface designers by now):

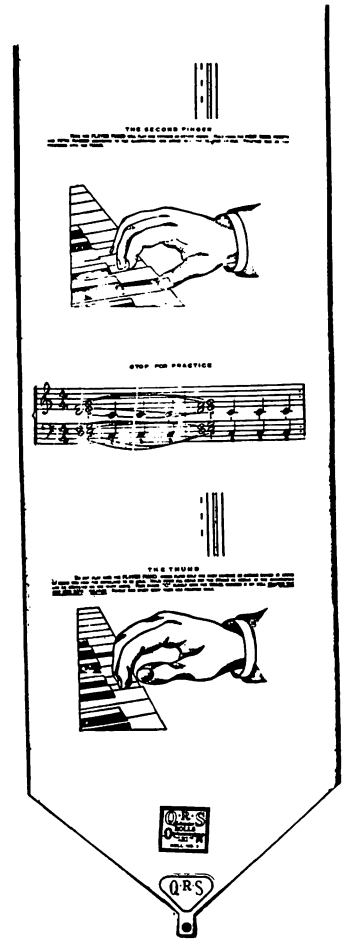
```
NY Times June 5, 1918  Carnegie Hall/Metropolitan Opera Symphony

Mr. Ornstein at first sat quietly on the stage while, in popular phrase,
"the piano played itself."
Two motors, as the instrument's inventor explained,
spurred a Knabe concert grand during the first movement of the concerto,
reproducing in power and speed, in all dynamic gradation and tonal shading,
the artist's interpretation.
The sense of personality in touch and tempo was only visibly modified,
a change in sight, not of hearing,
when the power was switched off and Ornstein himself took the keyboard
to finish the two remaining divisions of the concerto.
```

Into the 1940's most concert pianists made reproducing piano rolls instead of acoustical recordings. Rachmaninoff, for instance, had an exclusive contract with the Ampico company (after having had a bitter fallout with Thomas Edison) and some of his performances were only recorded on piano rolls (a notable one being his transcription of the *Star Spangled Banner*).

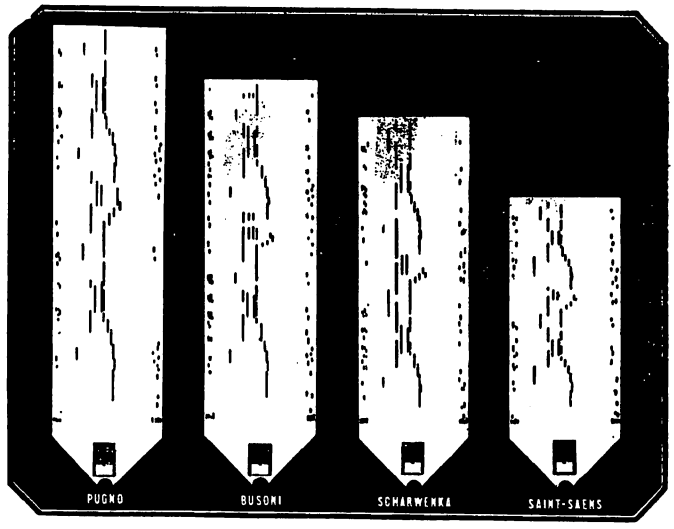
A few people are converting old rolls to MIDI floppy disks, but these are barely available. Someday old reproducing rolls will be a fascinating source of measurable historical data, something like Baroque barrel organs, which are our only recorded data of Baroque music actually being performed.

In some ways, the personal computer of today is like the player piano of yesterday - an expensive entertainment gadget, a middle class family devotes a corner of a room to the thing, and from time to time kids go out to buy the latest "program" to try out on their machine at home. Early on, manufacturers chased the computer-aided instruction and OEM software markets (without much success). The following ad was for a music roll that taught piano playing. (The roll was very expensive, $11.50 in 1923, and lost money for QRS):

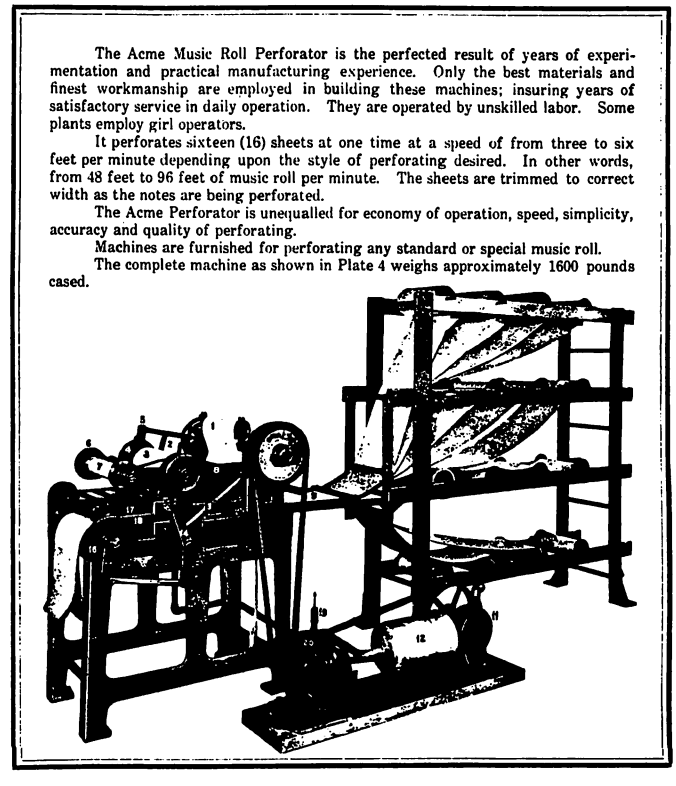




And this ad showed the consumer that temporal (rhythmic) data could be conveniently compared using graphics:



Budding programmers who wanted to work at machine-level could buy do-it-yourself perforators, from simple desktop models to great, hulking ACME perforators:



Finally, it is worth noting that, despite the availability of good synthesizers and pc's, the MIDI equivalent of piano rolls are not yet really available, and I haven't yet seen consumer applications that employ two devices that were well-used by piano roll makers: the *tempo line*, and lyrics. Song lyrics were printed in the right hand margin of many rolls so people could gather and sing along; and an ink vertical tempo line ran the length of the roll, abscissa giving the tempo. As the

music rolled by, the user tracked the tempo line with a needle connected to the speed control, to bring an otherwise mechanical performance to life. It is certainly within our power to implement rubber-band tempo lines for steering the performance of musical computers or speech synthesizers, and it is almost possible now to have computers track singing voices using fairly inexpensive pitch-to-MIDI converters that are becoming available in the consumer market. Considering how important it is, how vital to the expressivity and appeal of a message, rhythmic nuance in speech and animation as well as music, is remarkably poorly understood.

Where Things Stand

The Bösendorfer has just moved to the new Media Lab. John W. Amuedo is the principal scientist working with it. While Kimball are constructing additional Z-80 and IBM PC interfaces, we are now beginning to consider more powerful workstations and how to best employ the piano in the context of a larger computerized orchestra. Most immediately, we hope that at least some kind of workable data format can be made available soon so that others can analyze and compute Bösendorfer scores.

References

American Player Piano Supply Company Catalog, Wichita, KA.

Player Pianos and Music Boxes (Keys to a Musical Past)
Harvey N Roehl, Vestal Press, Vestal NY 1968
(Source for the old advertisements).

John Amuedo, personal communcation.

A Low Cost, Video Based, Animated Movie System for the Display of Time Dependent Modeling Results

William E. Johnston
Dennis E. Hall
Fritz Renema
David Robertson

Advanced Development Projects
Lawrence Berkeley Laboratory
University of California
Berkeley, California

ABSTRACT

A relatively low cost hardware and software, animated video movie making system is described. The system consists of an PC microcomputer based animation controller, video frame buffer and 1/2" editing video tape recorder. The characteristics and limitations of the system are discussed, together with the implications of encoding graphic primitives in an NTSC video signal. The use of the system for scientific movie making is described.

All of the authors can be reached via USMail at: Lawrence Berkeley Laboratory, Bldg. 50B, Rm. 3238, Berkeley, CA 94720. Email addresses are: wejohnston@lbl.arpa, ...ucbvax!lbl-csam.arpa!johnston; dehall@lbl.arpa, ...ucbvax!lbl-csam.arpa!hall; fritz@lbl-csam.arpa, ...ucbvax!lbl-csam.arpa!fritz (Currently a student at University of California, Berkeley, Electrical Engineering and Computer Science.); davidr@lbl-csam.arpa, ...ucbvax!lbl-csam.arpa!davidr (Currently a graduate student at San Francisco State University, Computer Science Department.)

The work presented in this paper is supported by the U.S. Department of Energy under contract DE-AC03-76SF00098. Any conclusions or opinions, or implied approval or disapproval of a company or product name are solely those of the authors and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory, or the U.S. Department of Energy.

Trademarks are acknowledged by †.

1. Introduction

We describe a system to simplify the display of time dependent scientific data by the use of an economical, video movie making system.

Our fundamental premise is that the process of doing video animation of the graphical output from numerical simulations is easy enough and inexpensive enough that it will have a significant impact on the way that the end user displays and explores time dependent modeling results. Video display permits easy handling of the graphics media for impromptu working discussions in the same way that paper display provides for single frame graphics. The equipment used to produce the video animation is inexpensive enough that individual researchers can afford to purchase and operate the system just as they would a graphics terminal. *The result is an increase in the use of movies for scientific data display, and new insights into the data because of this display methodology.*

2. Summary

The numerical modeling done on computers is frequently of time dependent processes. Examples are studies of the evolution of vortex structures in turbulent fluid flow, the diffusion of a gas or liquid through a porous medium, and the evolution of the shape of a particle beam in an accelerator. The output of these models is naturally displayed as a sequence of graphics frames in the form of a "movie", but the conventional process of doing this on a film recorder is a tedious and time consuming one, at best.

Our methodology consists of using a low cost, easily used, video tape based movie making workstation. The workstation consists of an IBM PC type microcomputer, a video animation controller, video image generation and display hardware, an editing video tape recorder, and a high speed communications interface. This video tape recorder (VTR) workstation is used directly by the end user, at his or her home site, to produce video movies of time dependent data. The sequence of graphics frames from time dependent numerical modeling is sent to the VTR workstation, there to be recorded in a standard, home video format. The video tape will be the user's working graphics output, much as, say, electrostatic plotter output has been in the past.

The workstation equipment costs about $10,000 in addition to an IBM-PC type of microcomputer.

3. Background

The use of computer controlled film recorders for the production of 16mm or 35mm film movies based on computer numerical simulations is tedious, expensive and time consuming. The film processing takes several days if there is no on-site color processing equipment, the editing phase requires equipment that is not commonly available, the film used in film recorders is not the type standardly used by commercial film movie makers (which causes reproduction problems), and the display requires a projector, screen and dark room. Compounding this for remotly located users is the necessity of relying on regular mail for access to the film recorder. *The result is that film movie making, while perhaps the ideal way to display the results of computer modeling, is seldom done, except for those sites that can afford local photolab, and substantial film recorder*

operational support.

An alternative to film based movie making is frame-at-a-time video taping. The techniques involved are similar to those used in video animation systems, where frame-at-a-time recording is done, as opposed to the real time recording that is done from video camera signals. The natural result of computer generated graphics is frame-at-a-time source material. In film based movie making this presents no special complications. In video recording, frame-at-a-time source is difficult to record owing to the lack of inexpensive, precise tape positioning hardware. The approach described here identifies what we believe to be the least expensive equipment that will provide the video animation capability.

4. Approach

The assumed scenario is that the numerical simulation generates a graphics metafile, or compressed raster images, of the graphics frames and sends that representation to a VTR workstation at the user's site. The volume of data to be sent may be large, but is not impossible to deal with. Precomputed raster graphic images usually compress well, both spatially (along scan lines), and temporally (from frame to frame), owing to high coherence in both dimensions. (See [1].) Alternatively, a high level metafile encoding can be sent and interpreted locally. The term "metafile" is used in a general sense, and could, for instance, be at a very high functional level (for example "plot contours").

Once in the VTR workstation at the user's site, the image is rendered in the video frame buffer that is attached to the workstation microcomputer, and recorded on video tape one frame at a time. (The term "render" is used to indicate the process of converting the metafile or compressed raster image into a bit mapped image in the frame buffer.) In general, the frames of the movie (animated sequence) cannot be rendered into the frame buffer at video rates (30 frames per second). This means that the VTR has to be able to record one frame and then wait a relatively long time for the next frame. This results in the requirement for the VTR to do single frame recording. This single frame recording is done by a process called "insert editing" on the VTR. An insert edit places the current video image (which has been stored as a single video frame in the frame buffer and is refreshed at video rates) at a specified frame location on the video tape. (The tape must be pre-formatted, with each frame uniquely numbered, and recorded with a black video signal.) For the next frame, a new image is rendered into the frame buffer, and the process repeated to add the new frame to the tape. The computer control of the VTR generally, and specifically the operation of insert edits needed for our video movie making is handled by an "animation controller". The animation controller presents a simple command interface to the computer (e.g. "editin at frame 2000 editout at frame 2002"), and interfaces to the control function signals of the VTR. The essential job of the animation controller is to provide the precise control of the VTR necessary to gate the video signal, and enable the edit-in function of the VTR, at exactly the correct time for a single frame edit.

The equipment required for VTR based animation has been available in the video industry for some time. (The term "video animation" system is applied generally to

frame-at-a-time video recording systems.) The problem with this equipment is that the quality of VTR, the video sync and time base generators, and the computer interfaces that have been used by the video industry for these video animation systems are designed to produce broadcast quality signals, and cost $50,000 to $75,000, minimum. The system described in this paper costs on the order of other graphics workstation equipment used by scientific graphics users; namely about $10,000, plus a suitable microcomputer.

A video industry system that provides simple VTR animation control and frame editing functions is called a "News Room" system. The design of our VTR workstation is based on this type of system. In our case, the images are sent from a back-end system in a variety of different formats. The software that runs in the VTR workstation will produce the images to be recorded by interpreting one of the several formats generated by the back-end systems. For instance, decompressing compressed raster images, or interpreting graphics metafile representations for vector graphics images. The image editing and titling functions of a "News Room" system are provided by PC software like "Dr. Halo†" or "Deluxe Paint†".

The resolution of video tape is limited in several ways to be discussed later. This limited resolution does not preclude useful video movies because the graphics representations that are useful for "movie" display of abstract figures are typically of low resolution. Since the type of graphic images under consideration are not similar to natural scenes, and therefore are not amenable to quick visual interpretation, the main things to be conveyed in a movie representation are the overall time dependent changes. With reasonably designed graphic displays, the important content should be able to be conveyed with the video movie format. Further, the video movie making process is quick and easy enough that adjustments to the movie are easily made to optimize the use of the available resolution.

4.1. The Animation Technique

The process of insert edit video animation is described below. Insert editing assumes that the video tape has been pre-formatted ("black-striped") with black video in the video frames, and with individual frame codes on an audio track. These frame codes (similar to the standard SMPTE time code) uniquely and sequentially label each video frame, and are used to locate specific frames on the tape for the edit operation. The animation algorithm is:

- Render One Frame:

  - Convert the graphics primitives which represent the image to pixels in the video frame buffer. The user interface is in how the graphics are presented to the workstation, and in the program needed to interpret the users representation and convert it to a frame buffer image.

- Record One Frame:

  - Use the frame codes to preform a single frame edit. The controller accepts the commands of the form "vcr edtp2; edin 2000; edot 2002; edit" to record, for

example, two copies of the video frame currently in the frame buffer onto tape, starting at frame 2000, in the insert edit mode.

- The VTR edit process may be represented on a time line:

```
S.O.P. |------------\\\----------------| E.O.P.
        ~5 sec    edit    ~4 sec
```

Referring to the above time line:

* Search to start of pre-roll (S.O.P.).

* Do the pre-roll. (The pre-roll is a period of time prior to the edit when the VTR is running at normal play speed. This is used as a settle time for the servos and tape transport, in order to ensure frame accurate positioning.)

* Perform the insert edit. (Re-record one or more specified frames on tape. This type of recording involves erasing exactly one frame (which is recorded as a diagonal stripe on the tape) and recording a new video signal on that frame. One of the things that differentiates editing recorders from others is their ability to erase a single frame. This implies that the erase head has to be on the rotating drum, along with the video record head, instead of being a transverse erase head that goes perpendicularly across the entire tape, as is the case in home video recorders.)

* Post roll.

* Stop.

• Repeat the Process.

Figure 1, below, illustrates the use of the video, control and audio tracks on the video tape for insert editing. The insert edit process done by the DiaQuest animation controller uses the time codes written into one of the audio channels. It does not use the control track which provides only frame markers, and no unique frame identification information.

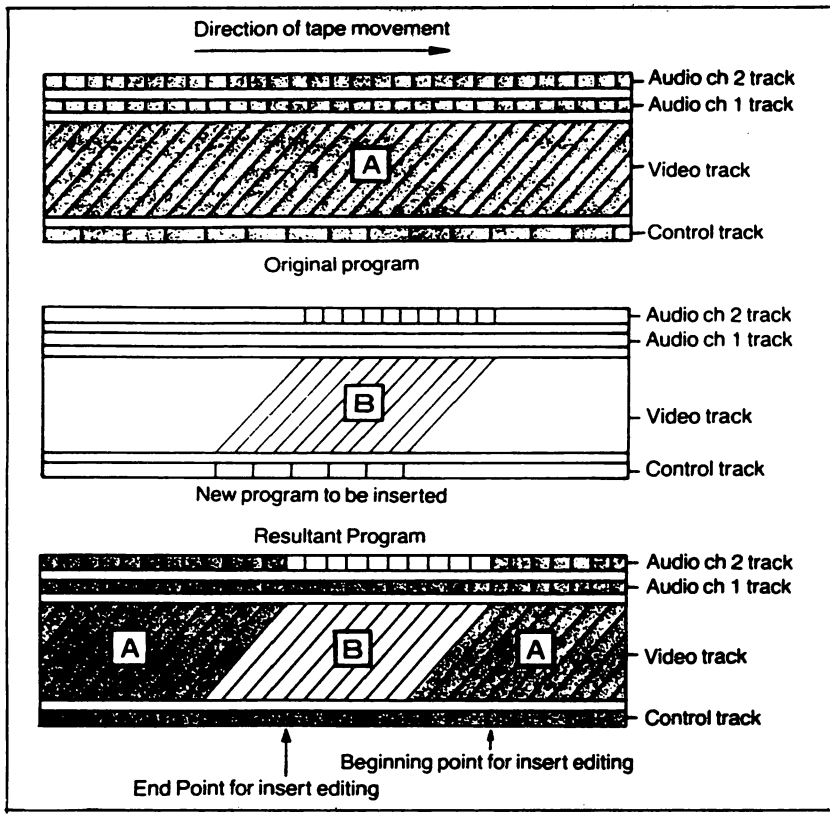


Figure 1 - An illustration of the insert editing process where tape B is the source and tape A is the target.

The necessity for relatively long pre-rolls is due to the transport geometry and the required accuracy for single frame edits. For example, in the case of a β-I format VTR, video tracks 0.1mm wide are being written at 275 ips, with the frame identifying information being read 7 inches linearly away from the start of the video frame record. The start of record timing must be accurate to about 5 microseconds which illustrates the problems of correct positioning and timing associated with the video tape transport. Figure 2 shows the transport geometry and the tape format for the Sony, SLO-338[†] 1/2 inch VTR used in our system. Recall that the frame identifying code is in one of the linear audio tracks.

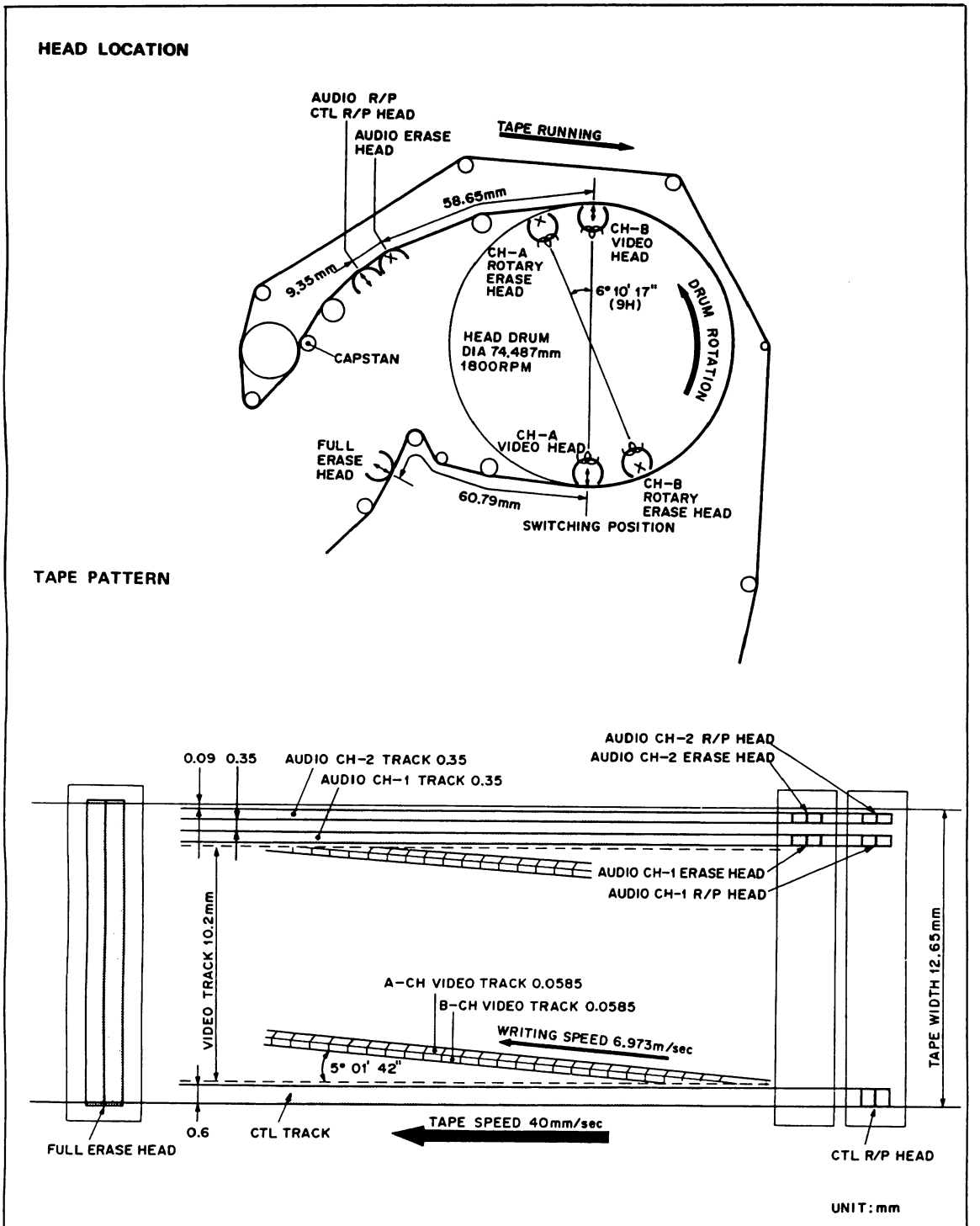


Figure 2 - The tape transport geometry and tape format for the SONY, SLO-383 VTR. (From ref. 2.)

4.2. VTR Workstation Overview

The logical configuration of hardware and software of the system are illustrated in Figure 3.

To briefly describe the components of the system as illustrated in Figure 3:

• The communication interface module manages the input of data from a back end system. The data input is via Ethernet using the DARPA, FTP protocol, the SUN Microsystems distributed file system (NFS†), or DEC's DECNET DOS†. Future plans include the use of remote procedure calls to both transfer data from the back-end system, and to control the operation of the workstation. The rationale for this is to introduce as few new user interfaces as possible.

• The image generation module interprets the graphics metafile or compressed raster image and converts those representations to calls to the graphics device driver for the attached frame buffer, and then generates the sequence of instructions needed to control the VTR for animation.

• The frame buffer provides a construction area for the image that is being rendered. The frame buffer also contains the hardware to convert the digital, pixel image into a video signal. The three channel Red-Green-Blue (RGB) video output of the frame buffer is converted into a single NTSC signal for the VTR.

• The image editing module provides the interactive ability to add titling or frame annotation to images in the frame buffer before they are recorded.

• The VTR animation controller provides for the basic VTR control functions (start, stop, pre-roll, edit, etc.), and tape formatting (black-striping).

• Black-striping is the process of formatting the video tape with black video and the individual frame codes that are used for single frame insert edits (the basis of animation). One of the important cost saving features of the particular animation controller that was chosen is that it contains the hardware to read video and sync information from the frame buffer, and generate corresponding time code for formatting the video tape.

• The VTR control module provides an interface to use the animation controller to manually control the VTR from the workstation computer keyboard. This is useful for searching and reviewing previously recorded material.

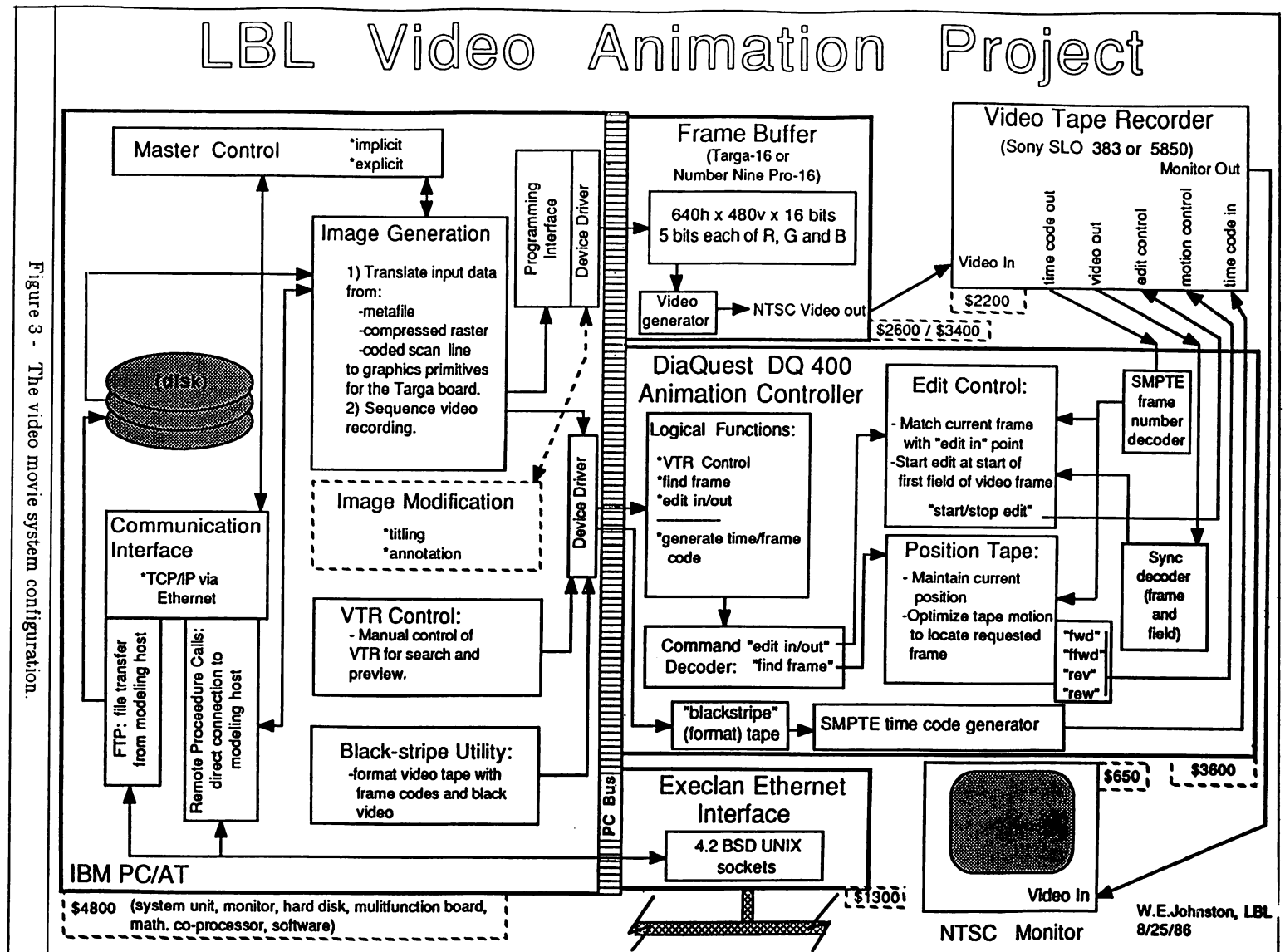


Figure 3 - The video movie system configuration.

4.3. VTR Workstation Characteristics

4.3.1. General

The graphics frames to be edited onto a tape can be in a variety of formats or types of compression. The rendering of the graphics metafile or compressed image into the PC frame buffer is a straightforward task that can partially overlap the edit operation. Furthermore, there is no particular necessity to use the PC to generate the video signal. Any NTSC signal source can be used, provided that it is frame buffer like (i.e., it preserves a single video frame for the ten seconds, or so, required to record), and that the source can signal the PC when the frame is ready for recording, and receive a signal to generate the next frame. (We know of one person who is going to use an Amiga† computer as the video signal source.) In this latter situation, the PC could be reduced to a minimal configuration (system board and floppy disk), as it's only function would be as a home for the animation controller board.

The spatial resolution of the recorded image is limited by the both the VTR, and the NTSC encoding, to be about 320h × 480v pixels. The color gamut of an NTSC signal is reasonable, and will probably be limited by the memory in the frame buffer where the image is rendered before recording. Anticipating that the graphics display techniques used for scientific data will increasingly be making effective use of continuous color (rather than pen plotter type discrete colors), the minimum color resolution that should be supported in the frame buffer is probably five bits per primary color (320 × 480 × 15 bits of memory for RGB).

The system is upgradable in the sense that, while the system being described recording workstation is low cost and based on a home video format, the same PC, software, and video signal can drive a professional editing recorder to produce master tapes that can then be used on a standard video editing system to produce an edited video program.

4.3.2. The NTSC Encoding and Computer Graphics Images

The single most significant characteristic of this technology is that the image to be recorded on video tape is encoded in an NTSC video signal. In brief, the NTSC encoding imposes a spatial frequency dependency on the color space gamut. In other words, the NTSC system separates a color image into three components which are encoded into signals of differing bandwidths. The color axes of the NTSC color space are Y (the monochrome luminance), I (the higher bandwidth color channel encoding a cyan-orange color axis), and Q (the lower bandwidth color channel encoding a magenta-green color axis). This means that the spatial resolution, or the largest number of resolvable horizontal dots or lines, is a function of the color being used. Quantitatively, this is expressed in the following table:

Table 1

Resolution of Color Components, NTSC compatible system

Type of color reproduction	Maximum video frequency, MHz	Maximum horizontal resolution, lines
Monochrome (Y signal only)	4.0	320
Orange-cyan (Y signal plus I signal)	1.5	120
Red-green-blue (Y signal plus I signal plus Q signal)	0.5	40

This implies, for instance, that if we try to draw line graphs with deep blue lines, that all the graphic information will be carried in about 10% of the signal bandwidth. The implications of this are that vector graphic images containing high spatial frequency line or dot patterns will reproduce well only in monochrome, and even then at a fairly limited resolution. Guidelines based on the above might be to select colors for line drawings from the high bandwidth color axis (cyan-orange), and avoid high frequency grids and dot patterns. Note that the resloutions given in Table 1 are the theoretical limits for the NTSC signal, and are not achievable by most, if any, video tape recorder. The principles of the NTSC encoding are discussed in greater detail in Appendix A.

These comments should not be taken in an entirely negative context, but rather a caution that the composition of images to be recorded in the manner being described requires attention to the system characteristics. We have made several very useful video movies which were particle and line representations using the video system which is being described.

4.4. VTR Workstation Implementation

The system is based on commercially available components for an IBM PC or compatible. Table 2 represents an existence proof, and lists the components of the system that we currently have running. The software components reflect the fact that we do as little software development work on the PC as possible, and that in a very plain, C language environment. The prices given are only for the purpose of quantifying our claim of "low cost". (The prices indicated are those available to large educational and government facilities. List prices are probably 10-15% higher.)

Except for the DiaQuest animation controller and the VTR that it will interface to, most of the choices for the rest of the hardware and software are a matter of taste or chance. We are planning to try a number of variations of the hardware components. Currently, much of the image rendering time is due to the PC having to generate all of the graphic primitives in software. A new generation of PC graphics boards like the Number Nine, Inc., 16 bit frame buffer board with the Texas Instruments graphics processor, should help this situation.

One of the common criticisms of our system by video people is that the SLO-383† VTR will not stand up the constant tape motion that our scheme implies. Each insert

edit involves playing $\sim$ 10 seconds, or about 300 frames, of tape. This means that each frame of the movie passes under the recording video head 300 times. So far our experience has not indicated any problems. We expect that periodic preventive maintenance of the VTR should produce acceptable results, and we use high quality tapes that are typically only used once (that is, for recording only one movie). Even if the VTR lifetime is shorter than normal, the SLO-383† is one of the less expensive components of the system and periodic replacement adds a fairly small cost to the operation of the system.

The Sony VO-5850 is a common, industrial, 3/4" VTR, and the control signal interface for the SLO-383† and the VO-5850† are the same, permitting the DiaQuest animation controller will run either one. The VO-5850† costs about three times what the SLO-383† does, and will probably last somewhat longer. An additional advantage is having your master tapes in 3/4" format which are somewhat higher bandwidth than the 1/2 inch β-I tapes.

For anyone who needs higher quality recordings the DiaQuest, DQ-422† controller has the same software interface and will electrically interface to the Sony line of broadcast VTRs, in particular the Betacam† recorder.

In practice what is frequently done is that the master tapes that are produced on the SLO-383† are dubbed to VHS for convenience. We use a Panasonic, NV-8950† for this purpose, though any high quality recorder will work. An assemble edit feature (which allows video material to be added to the end of the tape without a video "glitch" between clips) on the dubbing machine is very handy for piecing together clips from master tapes. Home recorders that are designed for video camera input usually have this feature. (A second SLO-383† and the matching RM-400 editing controller console do an excellent job of assembling clips into a high quality β copy.) The SLO-383† is a β-I format system, and while home β format recorders cannot record this format, they can play it back.

Table 2
LBL Video Animation System - Approximate Costs

Item		Required for PC	Required for video system	Optional
IBM PC/AT (or clone)				
system unit w/1.2MB floppy				
and 256KB memory	2700			
math co-processor	180			
B&W monitor	190			
Hercules monochrome board	280			
AST† multi-function board				
w/1MB memory	490			
41 MB hard disk	875			
(total)	4715	4715		
PC/AT Software				
DOS 3.1	60			
Fastback†	94			
Micro-Soft C	204			
Assembler	77			
On Track disk manager	100			
Polytron Make	96			
PC-VI† editor	75			
(total)	706	706		
Sony, SLO-383† editing video recorder			2110	
AT&T, Targa-16† frame buffer (w/over scan)			2514	
Sony, PVM 1271† monitor			730	
DiaQuest, DQ-400† animation controller (One of the key elements of the movie system is this animation controller which is sold by GESI, 1440 San Pablo Av., Berkeley, CA 94702. Ph: 415/527-7700)			3600	
Execlan Ethernet board w/socket library			1295	
Simple image editor				
Dr. Halo†	495			
mouse	122			
(total)	617			617
Sony, PVM 1910† monitor				677
VHS dubbing video recorder (Panasonic, NV-8950†)				1500
Total ($)		5421	10206	2847

4.5. Programming Considerations

The software development for this project has been minimal. The programming interface to the animation controller is trivial. It involves writing a few simple, human readable commands sent to the PC's "COM1" port, where the DiaQuest device driver intercepts the ASCII strings and sends them to the command parser on the animation controller board. A typical video movie making program reads a metafile which has been sent to the PC using FTP. A second file is used to keep track of the current position in the animated sequence, so that sequence may be made up of "clips" which are

represented by convenient size metafiles. This second file is also used to communicate a few editing commands to the program: How many times to repeat the first and last frames of the movie, and how many instances of each graphic frame is to be recorded on tape to achieve animation (two is typical). To help avoid confusion, the metafiles may contain the frame number within the movie of the first frame of the metafile, so that sequencing can be checked if the movies are made up of several clips, as is typical.

In fact any reasonable graphics representation may be used. It is frequently easier to adapt programs on the PC to accept whatever is generated on the computer doing the modeling, than to try to change those modeling codes. This is especially true of the very large programs that run on supercomputer backend systems. This later circumstance, for example, lead us to write a Tektronix 4010 code interpreter for the VTR workstation. We have made several useful movies using Tektronix code because that is what is generated by the only graphics package available on a particular supercomputer used by researchers at our institution.

4.6. Experience, Timing, Etc.

Of the video movies that we have made so far, most have been fairly simple line drawings or particle scatter plots. To site a specific example, our first movie was of a collection of particles being advected through an turbulent flow field. There were about 800 distinct frames, each of which was recorded twice in the movie. Each frame contained from 1000 to about 5000 particles, and took about 10-15 seconds to draw into the PC frame buffer. The movie was recorded in 10 clips. These 10 clips were contained in metafiles totaling about 40 Megabytes. The recording time, including transferring the metafiles from a VAX† to the PC, was about 10 hours. This double frame animated video movie plays for $800 \times 2 \times (1/30) = 53.3$ seconds, plus the first and last frames which are held for about 20 seconds each. To put the movie making process in perspective, the modeling of the flow field took about 30 hours of Cray X-MP time, and the particle advection took several hours on a VAX† 780 UNIX† system which also generated a simple metafile representing the particle positions in time.

5. Future Development

First, antialiasing should be done. The line drawings suffer from the "jaggies", and we have considerable color resolution available to correct this. Second, we are now working on 3D problems that are going to require some surface rendering for the display. In both cases (because the PC is very slow) we will probably want to generate raster images on a VAX† (or maybe a Cray) and send them down to the PC. Compression will, of course, be essential for this, and the block truncation coding (BTC) algorithm [1] should work very nicely. Additionally, we would like to be able to use the PC as a preview device. In order to get reasonable viewing speed it may be possible to use a hardware implementation of the BTC decoder. We are investigating this possibility with Frederiksen and Shu Laboratories.

Specifically, we working on the graphical display of 3D flow fields. The display will be of advected particles and streamlines with the relevant boundaries shown. To do this we are putting together some z-buffer tools which we will run on a VAX†. The images will be generated on the VAX† at high resolution, averaged down, compressed and sent

to the PC for recording. (The reason for the z-buffer is that another project involves display of parametrically defined surfaces which we want to reneder directly into the z-buffer without tessellation.)

6. Acknowledgments

The authors would like to thankfully acknowledge Vincent Casalaina, a free lance producer/director, and instructor of videography in the Graduate School of Journalism, UC Berkeley. Vincent Casalaina provided much thoughtful input early in the conceptual phase of the project, reinforcing our belief (contrary to the advice of several professional video engineers) that what we wanted to do was possible. Thanks also goes to Al Brewster of GESI, Berkeley, CA, for providing technical expertise in our use of the animation controller.

7. Appendix A: The NTSC Encoding - Principles and Practice

The single most significant characteristic of this technology is that the image to be recorded is encoded in an NTSC video signal. The NTSC encoding is the result of requiring that color video signals be transmitted in the same broadcast spectrum bandwidth slot as was the original black and white video signal (about 4.5 MHz for the video portion of the signal), where one might reasonably expect that three times the bandwidth would be required.

The approach taken in this presentation will be an essentially historical one, as that is how the technology evolved. We first discuss signal compression due to separation of luminance and color information, then by matching the bandwidth of the color information to the responce of the human eye, and finally by signal processing techniques which allow the transmission of the color information interleaved in the luminance signal.

The necessary color signal bandwidth compression is accomplished by the NTSC encoding, which mainly involves two techniques. One of these techniques discards insignificant information, the other may discard significant information. The first technique results from the physiological characteristics of the human visual system. The eye exhibits the tritanopic version of dichromatism for objects of "medium to high" spatial frequency. This means that, for objects in the range of about 12 to 20 minutes of arc, that the human eye matches all spectral colors with only two primary colors instead of the usual three (dichromaticity). For tritanopism the primaries are 578 mμ (yellow) and 410 mμ (violet). (The term tritanopic comes from optometry, where it is used to describe color blindness of the same dichromaic type, but for all spatial frequencies, instead of just high frequencies.) Under these circumstances of small subtended angles, the luminosity response of the eye is approximately normal, but that hue discrimination is much poorer than normal. Visual scene objects generating these medium to high spatial frequencies subtend an angle of between 10 and 25 minutes of arc. (A US 5-cent piece subtends 15 minutes of arc at about 15 feet from the observer). For high spatial frequencies the best discrimination is in the orange-red region of the human color gamut. For even higher spatial frequencies the eye perceives monochrome luminance only.

The relationship between video signal bandwidth and object size is that the luminance variation caused by a small object (small relative to the length of the scanline

across the scene) are represented by high frequency video signals. The standard scanline is about 53μus long. An object which covers 1/640 of that scanline (e.g. one pixel of a typical frame buffer) generated frequencies of

$$\frac{1}{\left(\frac{53\times10^{-6}\mu s}{640}\right)} \approx 12\ MHz$$

The first part of the above observations (namely, reduced color perception at high spatial frequencies) leads to a signal compression technique known as the "principle of mixed highs". In essence, mixed highs involves converting a full bandwidth (RGB) color signal to four, 2 MHz bandwidth channels. Three reduced bandwidth color signals (0-2 MHz each) are used to faithfully reproduce low spatial frequency colored areas, and a luminance channel is used to deal with high spatial frequencies (2-4 MHz). Figure 4 shows a signal processing block diagram of the mixed highs principle, and Figure 5 illustrates the principle visually.

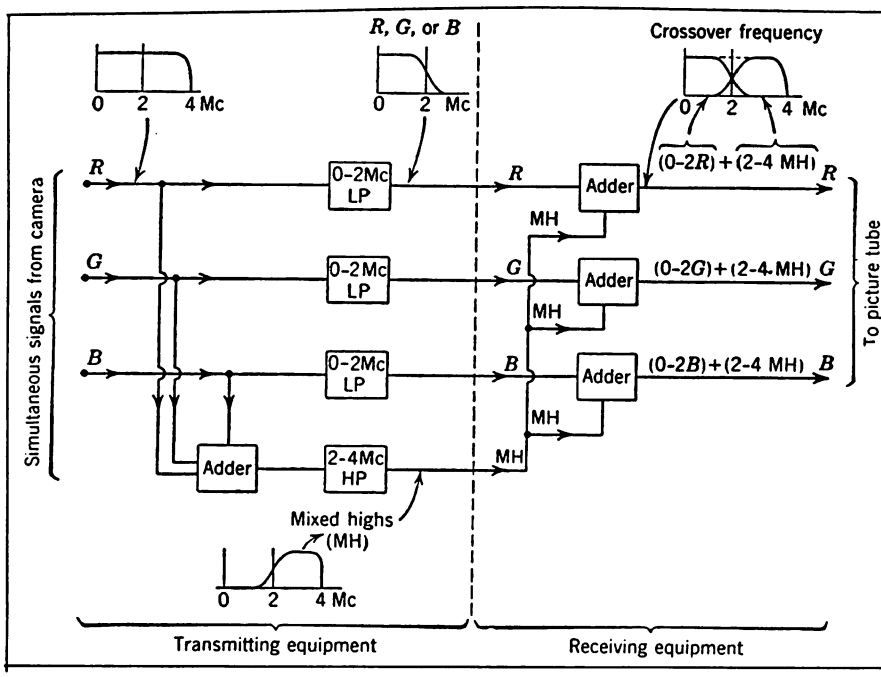


Figure 4 - Block diagram showing a mechanism of constructing a mixed-highs signal. (From Hazeltine, ref. 4.)

Figure 5 - A visual illustration of the principle of "mixed highs", namely that a low bandwidth color signal added to a high bandwidth black and white signal will give good color reproduction. The upper left is the high bandwidth monochrome image, the upper right the low bandwidth color image, and the lower image is the result of mixing the other two. (From Evans, ref. 6.)

Further compression of the signal (now reduced from 12 MHz to 8 MHz) can be obtained by noting that at medium to high spatial frequencies, the eye does not have a uniform color resolution (color hue discrimination ability). The tritanopia exhibited by the eye at high spatial frequencies is essentially a statement that the eye has high and a low bandwidth color signal channels. In particular, in the dichromatic sensitivity region, the observation is made that hue discrimination for red-orange is 7 mμ for "just noticeable differences" (JND) between color hues, while for other colors the JND goes up to 48 mμ. Figure 6 shows a measure of constant JND's for a normal eye and normal sized objects. It is clear from the orientation of the JND ellipses that the tritanopia of high spatial frequencies has precursors in normal vision.

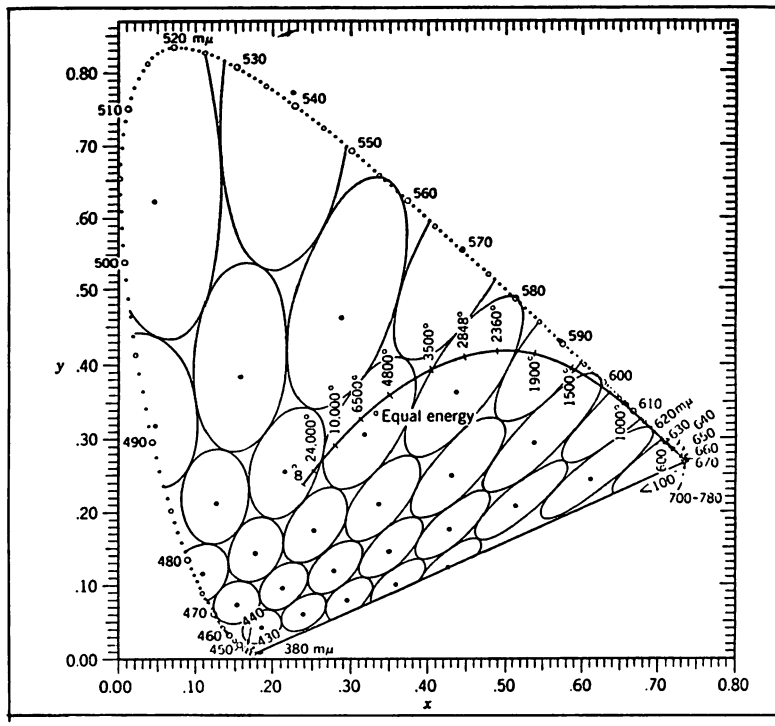


Figure 6 - Approximate perceptibility of chromaticity differences on the *xy* chromaticity diagram. Each closed curve shows 100 time the JND under moderately good viewing conditions. (From Judd and Wyszecki, ref. 7)

The tritanopia of the normal eye for small objects causes confusion between colors lying on lines which radiate generally upward from the blue corner of the CIE chromaticity diagram. Figure 7 shows the loci of poor color discrimination at high spatial frequencies. Figure 8 shows the relationship of perceived colors to the CIE diagram.

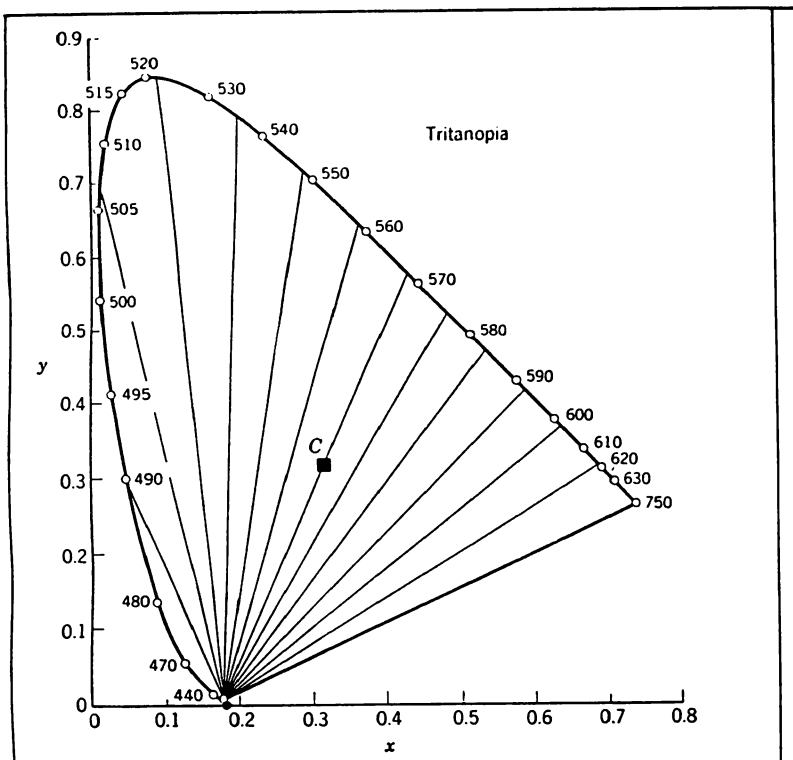


Figure 7 - Loci of colors confused by tritanopic color blindness. (From Hazeltine, ref. 4.)

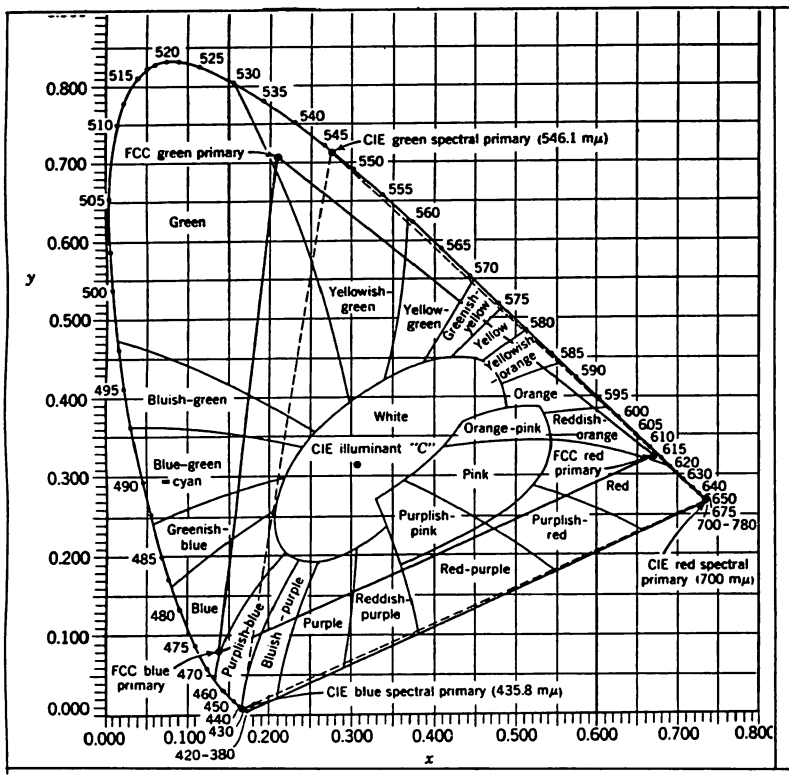


Figure 8 - Chromaticity diagram with color designations of various areas. (From Hazeltine, ref. 4.)

This line (the tritanopic locus through the equal luminance point "C") can be used as the axis of colors for a narrow bandwidth color channel, while the "orthogonal" axis defines the high bandwidth channel. These color axes of minimum and maximum spatial

frequency response are illustrated in Figure 9.

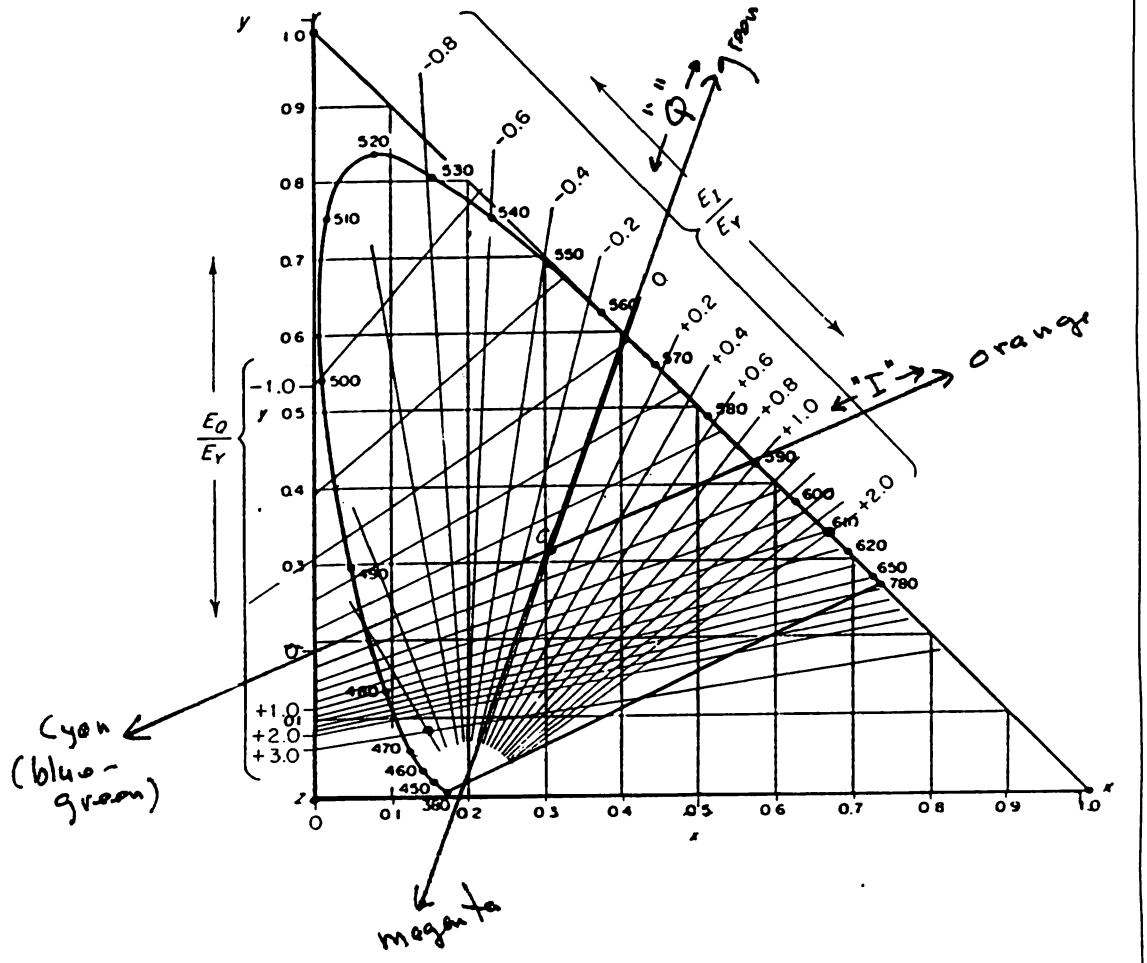


Figure 9 - The NTSC I and Q color axes shown in relationship to the CIE, *xy* diagram.

The narrow band color channel chromaticities go roughly from magenta to green (and are referred to as the "Q" axis in NTSC terminology), while the wide band channel chromaticities go from orange to cyan (the NTSC, "I" channel). For signal transmission reasons it is desirable to represent the color signals as differences from the luminance signal such that these differences vanish when gray (achromatic) color is being represented. The resulting color representation is called a YIQ color space, where Y is luminance, and I and Q are color difference signals with the characteristics described above. The

channel bandwidths of these signals are approximately 4 MHz, 1.3 MHz, and 0.5 MHz, respectively. To elaborate slightly on the implications of the differing bandwidth color signal channels, large colored areas (areas of low spatial frequency) imply only low video signal frequencies (f< 0.5 MHz) and have full color reproduction. In this range all three signal components are present. Picture areas having spatial detail in the range of 21 to 8 minutes of arc (as seen at a distance of four times the picture height), correspond to video signal frequencies in the range 0.5 MHz $\leq$ f $\leq$ 1.3 MHz, in which case the Q channel signal is absent, and color detail is reproduced with two colors, that is I (cyan-orange), and Y (luminance). For areas with detail of less than 8 minutes of arc, the video frequencies are above 1.3 MHz, both the Q and I signals are missing, and reproduction is monochrome (luminance only). Figure 10 shows the bandwidths of the Y, I and Q signals, and their placement in the NTSC modulation signal. Figure 11 visually illustrates the relative bandwidths of the Y, I and Q signals.

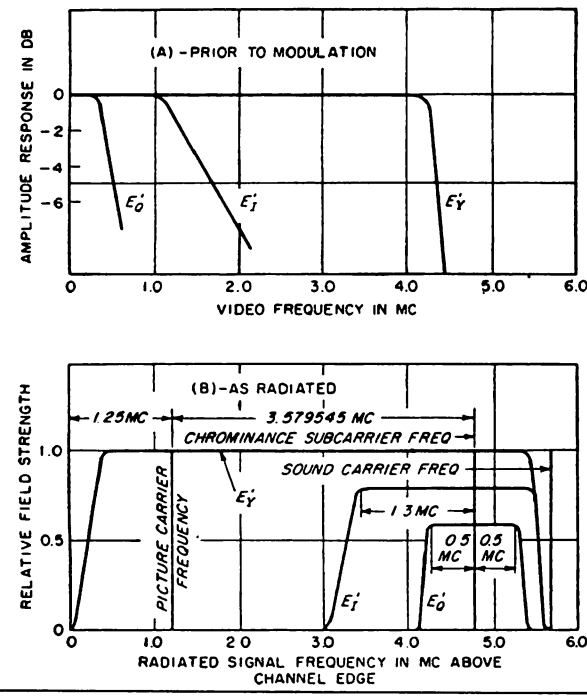


Figure 10 - Relatvie bandwidths of the luminance and chrominance NTSC signals:top' vidéo signals prior to modulation; bottom, after modulation. Relative amplitudes not to scale. (From Fink, ref 3.)

Both of the latter two cases are consistent with how the eye perceives color. By way of comparison, in a 640×480 display typical of computer graphic systems claiming compatibility with broadcast television signals, pixels are 1.5 minutes of arc wide at the 4 × screen_height viewing distance. The 21 to 8 minutes of arc feature size that is reproduced by I and Y corresponds to 14 to 5.3 pixels on a 480H × 640W display, and features smaller than 4 or 5 pixels will be reproduced in monochrome.

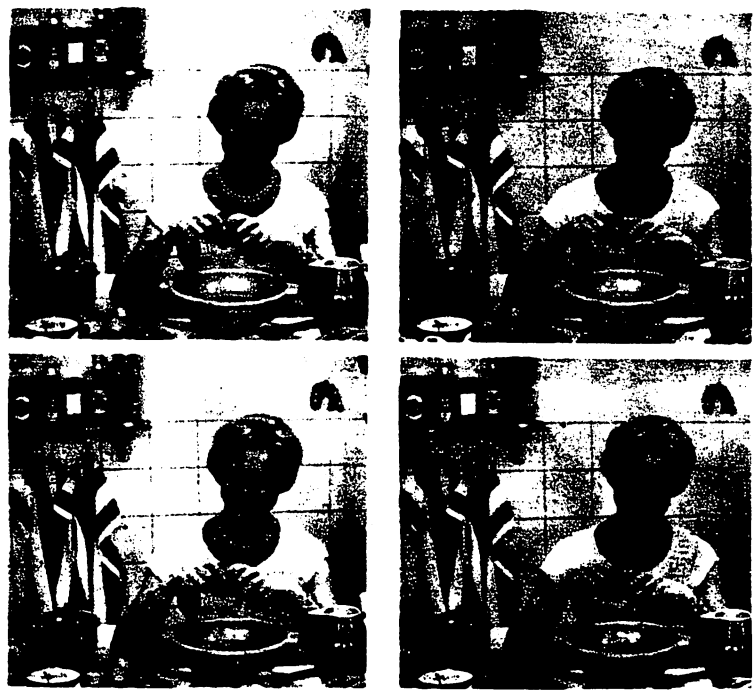


Figure 11 - An illustration of the relative bandwidths of the NTSC, Y, I and Q signals. The image in the upper left is just the Y component, the upper right the I+Y component, the lower left the Q+Y, and the lower right the Y+I+Q components. Note that the color contrast is a measure of the bandwidth, and that the Q+Y image is a low contrast, bluish image. (From Hunt, ref. 5.)

We now have a total signal bandwidth of $4.0 + 1.3 + 0.5 = 5.8$ MHz. This must still be reduced to about 4.5 MHz for broadcast. The further bandwidth reduction is obtained by interleaving the chrominance signal with the high frequency luminance signal. This is mostly a signal processing issue, and involves using parts of the video signal spectrum that are not used by *normal* video material. The method of encoding the chrominance signal (on a high frequency subcarrier) together with the luminance is very interesting, but somewhat beyond the scope of this discussion. (See the references mentioned below.) Figure 12 shows how the luminance and color signals are interleaved.

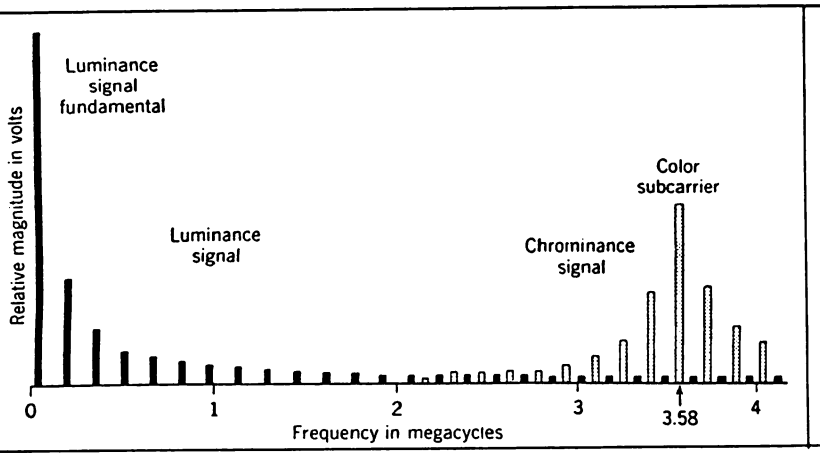


Figure 12 - Transmission spectrum for an NTSC video signal showing the interleaved luminance and chrominance frequence components. (From Hazeltine, ref. 4.)

Figure 13 shows the video signal for a flat, yellow field, and Figure 14 shows the spectrum of that signal.

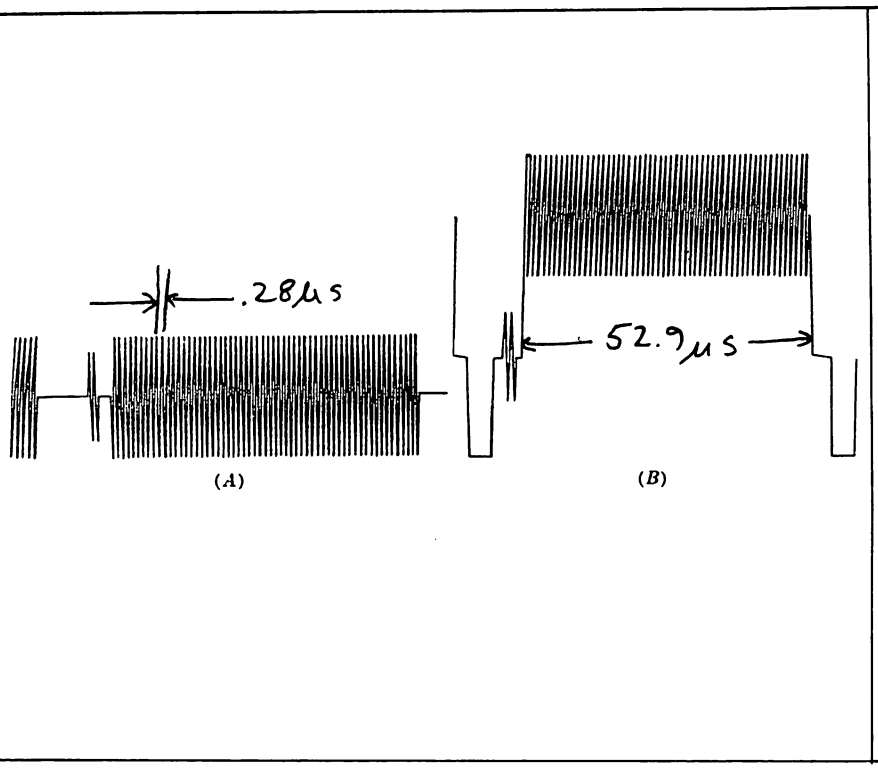


Figure 13 - (A) The color burst sync signal, and the chrominance signal as an amplitude modulated color sub-carrier. (B) A single scanline of the video signal of a flat, colored area, showing the luminance and chrominance components. The period of the color sub-carrier is indicated, together with the length of the scanline. (From Fink, ref. 3.)

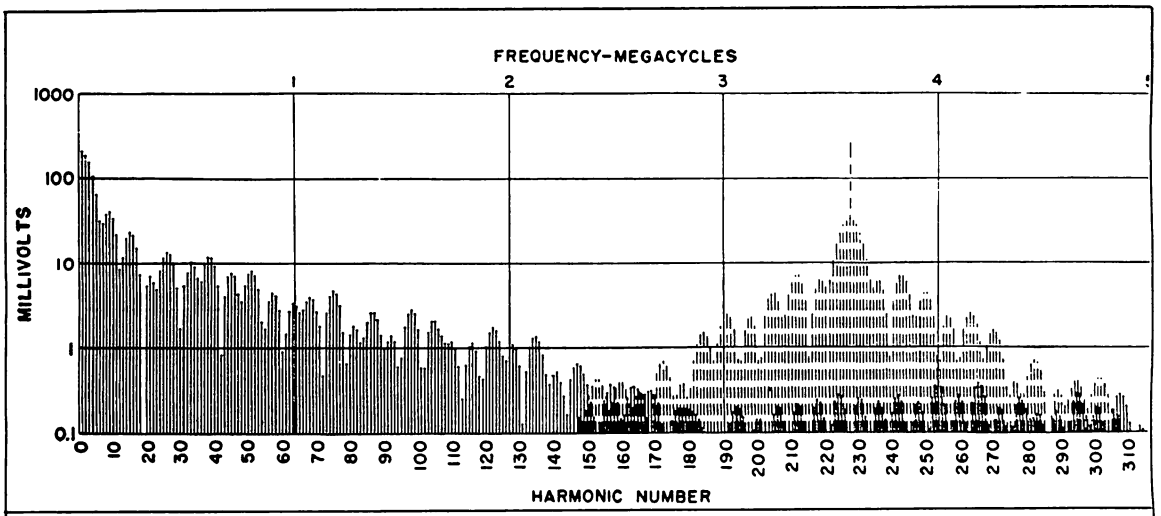


Figure 14 - The frequency components of the mixed luminance and chrominance signal shown in Figure 13 (B). (From Fink, ref. 3.)

Note that the flat yellow field signal is no different from a high spatial frequency monochrome dot pattern. Given an active scan line of 52.9μs, the color sub-carrier period, in terms of pixels, is

$$\frac{640\ pixels/scan\_line}{\left(\frac{52.9\mu\ sec/scan\_line}{0.28\mu\ sec/color\_sub\text{-}carrier\_cycle}\right)} \approx 3.4\ pixels$$

In other words, a pattern of non-unity luminance pixels on 3 pixel centers looks very much like the NTSC representation of a flat yellow field. One of the ways that color information is kept separate from high frequency luminance information is that the phase of the color sub-carrier is inverted from frame to frame due to the color sub-carrier being an odd multiple of the line frequency. This does not affect the color information, but as a luminance signal this tends to cancel itself. The signal interleaving works because in a typical video signal rapid luminance transitions are rare, and can be distinguished from the high frequency color sub-carrier. For non-typical video source material (such as a dither pattern, for instance) the above assumptions are violated, and you have substantial luminance frequency components in the region of the color sub-carrier. This makes proper decoding of the NTSC signal difficult, if not impossible, resulting in erroneously decoded images.

The implications of the above discussion are that one cannot represent the arbitrary color images in an NTSC signal that can be represented in a set of RGB signals. In some cases the information we are attempting to represent in the RGB signal is meaningless (because it cannot be perceived by the eye), but in other cases the problem is that we are trying to represent images that are too different from those of the natural, macro-scale, world on which the NTSC encoding is based. Guidelines based on the above might be to select colors for line drawings from the high bandwidth color axis (cyan-orange), and avoid high frequency grids and dot patterns.

The following table gives the maximum attainable resolution for the three signal channels described above. These maximum resolutions stem from FCC regulations

limiting the channel bandwidths so that the broadcast frequency spectrum channel bandwidth is not exceeded.

Table 3

Resolution of Color Components, NTSC compatible system

Type of color reproduction	Maximum video frequency, MHz	Maximum horizontal resolution, lines
Monochrome (Y signal only)	4.0	320
Orange-cyan (Y signal plus I signal)	1.5	120
Red-green-blue (Y signal plus I signal plus Q signal)	0.5	40

Most of the material in this section was derived from [3], and [4], where further discussion may be found.

8. Bibliography

[1]
"Two Bit/Pixel Full Color Encoding,"
Campbell, G., T. DeFanti, J. Frederiksen, S. Joyce, L. Leske, J. Lindbergand D. Sandin, *Computer Graphics* **20**(4) (1986). (Proceedings ACM SIGGRAPH, 1986)

[2]
"SLO-383 Service Manual,"
Sony Corporation, Number 9-966-606-02 ().

[3]
"Television Engineering Handbook,"
Fink, D., McGraw-Hill (1957).

[4]
"Principles of Color Television,"
Hazeltine Laboratories Staff, John Wiley (1954).

[5]
"The Reproduction of Colour in Photography, Printing and Television, Third Edition,"
Hunt, R. W. G., John Wiley (1975).

[6]
"An Introduction To Color,"
Evans, R.M., John Wiley (1948).

[7]
"Color in Business, Science and Industry,"
Judd, D.B., G. Wyszecki, John Wiley (1975).

Object Oriented Programming in NeWS

Owen M. Densmore

Sun Microsystems
Mt. View, California

Abstract

The NeWS† window system provides the *primitives* needed to create window managers and user-interface toolkits, but does not, itself, supply either. This is done to achieve a layering strategy for building several higher level systems that can share NeWS as their low level window system. None of the traditional "tool kit" solutions currently span the diverse set of clients NeWS needed to serve; they simply lack sufficient flexibility. We are exploring an object oriented approach which uses a flexible inheritance scheme. This paper presents our initial attempt at introducing a Smalltalk style class mechanism to PostScript‡, and our first use of it.

Introduction to NeWS

NeWS is a server-based window system which replaces the usual network protocols for expressing window and graphics primitives by an interpreted programming language. The language consists of almost all of Adobe System's PostScript [5], with some extensions.

The extensions to PostScript include:

- Primitives for managing client TCP/IP style connections.
- Primitives for light-weight processes.
- Multiple drawing surfaces called "canvases".
- An event mechanism for handling user input and inter-process communication.
- Use of garbage collection.

See the NeWS Preliminary Technical Overview [4] for details. For the rest of the discussion, use of the word "PostScript" will include these extensions.

The NeWS environment consists of the NeWS window server communicating with client programs using standard TCP/IP. The client does not, however, have to use a fixed protocol for window and graphics primitives. Instead, each client connection has its own "private" light-weight process [LWP in the figure] executing the client's PostScript commands. This, in effect, replaces a network window protocol with a language.

† NeWS is a trademark of Sun Microsystems Inc.

‡ PostScript is a trademark of Adobe Systems Inc.

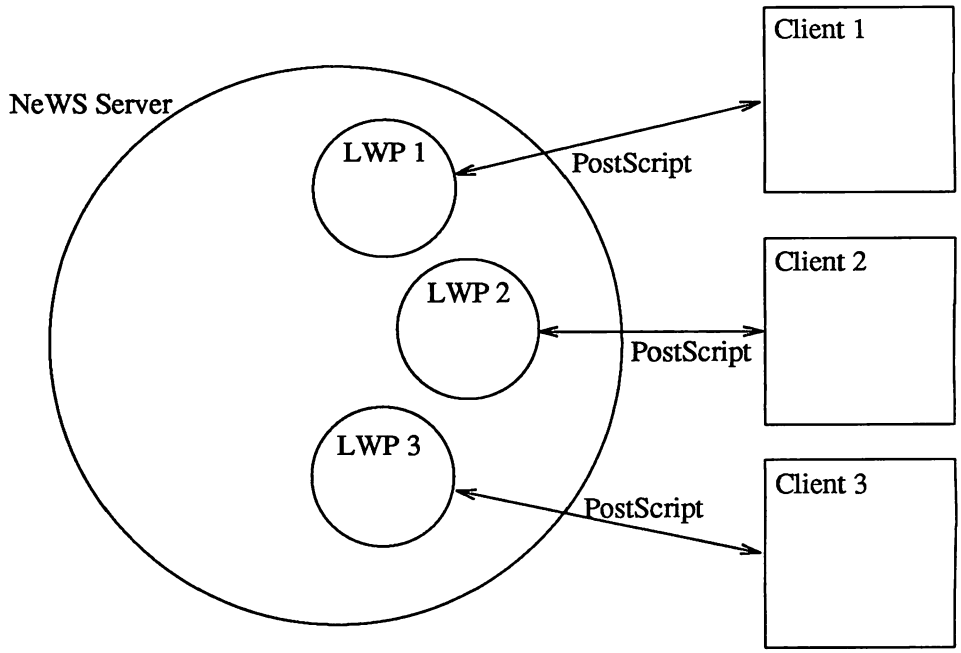


Figure 1 *NeWS Client-Server Environment*

Evolution of NeWS Tools

NeWS provides a rich set of graphics and window manipulation primitives but does not, however, provide window managers or user-interface toolkits. (NeWS is a "window kernel", not a "window shell".) This is done to achieve a layering strategy for building several higher level systems that can share NeWS as their low level window system. A major problem in providing this layer is the extreme diversity of the NeWS user community: OEM's, the Lisp community, our past SunView clients and X window system clients, to name a few. It became clear that the more traditional user-interface "tool kit" would have difficulty providing sufficient flexibility.

A second problem was *where* to put the higher layer – on the server or in the client. There are several advantages to providing the tools on the server side of the connection:

- Interactive performance would be improved since the user interface code would be kept "hot" in the server.
- Client code size would be decreased by sharing the server process' code.
- Since the NeWS environment is much more powerful than the typical C client's, it would allow very rapid prototyping. We show below, for example, that Smalltalk-like classes can be implemented in two pages of PostScript.
- No preference would be given to C clients over others; PostScript would be a common language for all clients. Whenever a new package was made available [by *any* client], it would immediately be available to all clients.

Our first attempt was to define a *package* to be a PostScript dictionary that managed an *object* that was also a dictionary. (Dictionaries are discussed below.) This met with success during our alpha release, especially in terms of flexibility. One Lisp client, for example, built a compiler to translate Lisp window calls into PostScript using the packages delivered with the system. Where they needed to modify the behavior of one of the package procedures, they replaced it with a procedure that called the original one.

This led us to look for a formalization of this style. A Smalltalk-like class mechanism seemed to fill this need. Just before our beta release, therefore, we decided to look for the extensions we would need to make to PostScript to support classes. Much to our surprise, PostScript could implement classes with no modifications! The secret is PostScript dictionaries.

PostScript and Dictionaries

PostScript is a forth-like (prefix notation, stack based) interpretive language developed by Adobe Systems [5]. It is a strongly, but dynamically, typed language. By *dynamically* I mean that object type is determined at run time. It is "polymorphic" in that an object may have different types at different times during execution of a program. In the figure below, for example, *foo* is assigned the number 10 in one context, and the string *abc* in another.

Dictionaries are compound PostScript objects that contain key-value pairs. They can be used in two basic ways. First, their values may be set and retrieved explicitly by the *get* and *put* primitives:

```
MyDict /foo 10 put ..causes the value 10 to be associated with the key foo.
MyDict /foo get    ..causes the value for the key foo to be put on the operand stack.
```

Second, their values may be set and retrieved emplicitly by use of the dictionary stack. Simply using a name in PostScript causes that name to be looked up in the set of dictionaries currently on the dictionary stack. The primitive *def* will define a key-value pair in the topmost dictionary, while the primitive *store* will first look to see if the key is defined in the dictionary stack, assigning it to that value if present.

This second form is the basis of PostScript's name scoping and over-ride mechanism. The set of names (primitives and data) known to the interpreter is the set of names in the current dictionary stack. A name can be re-defined by simply defining it in a higher dictionary.

The figure below shows three clients' dictionary stacks. All clients have a shared system dictionary and a private user dictionary. Two clients have additional dictionaries on their stacks. (Note: take care not to confuse the dictionary stack with the operand stack which has the operands and results of PostScript operators.) The system dictionary has a name *foo* which was defined at startup. Clients 1 and 3 have not redefined *foo*, thus will share the initial definition of *foo* (as the string "abc"). Client 2, on the other hand, has defined *foo* to be the integer 10. It is this capability to over-ride which makes PostScript adaptable to inheritance schemes in general, and to the class mechanism in particular.

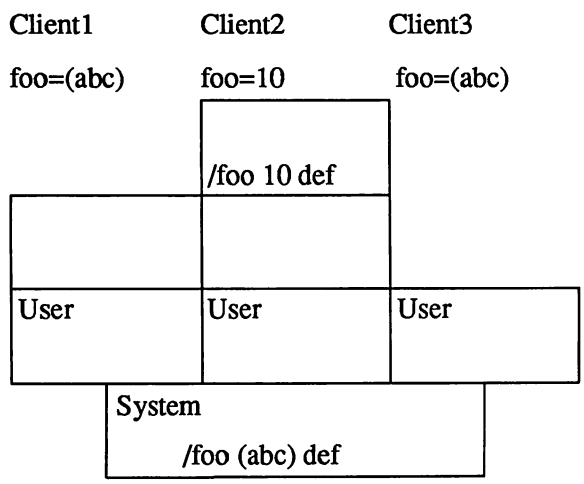


Figure 2 *Multiple Dictionary Stacks*

Modules and Classes

The reader unfamiliar with the use of message-passing, classes, and object-oriented programming may browse through the references listed at the end of the chapter [1, 2, 3]. Many of the essential ideas in class-based systems are similar to the more traditional "package" or "module" based systems, however.

Briefly:

- Packages (modules) are replaced by *classes.*
- Procedures in packages are replaced by *methods* in classes.
- Creating package objects is replaced by creating new *instances* of a class.
- Package local and global variables are replaced by *class variables.*
- Object variables are replaced by *instance variables.*

New notions are:

- Classes are ordered into a hierarchy by *subclassing* a new class to a prior one, *inheriting* its methods, instance variables, and class variables.
- Methods are invoked by use of the *send* primitive. The term *message* is used for an invocation of a method with its arguments.
- There is a means of constructing classes that is absent in most languages' module creation.
- Two new concepts, the *self* and *super* pseudo-variables are introduced. They are used in methods to refer to the object sent the method, and the method's superclass, respectively. Note: self does *not* refer to the method's class, but rather to the object that originally caused the method to be invoked.
- Unlike PostScript procedures, methods are *compiled* when a class is created. Currently this simply resolves self and super, and performs some minor optimizations.

The relationship between an instance and its class and superclass is shown in the figure below. We have made an instance, *aFoo*, of class *Foo* which is a subclass of class *Object.* An instance has a copy of all instance variables of its superclasses, thus *aFoo* has those required by both *Foo* and *Object.* The methods known by an instance are stored in the classes in its superclass chain. Thus *aFoo* can only respond to methods residing in *Foo* and *Object.*

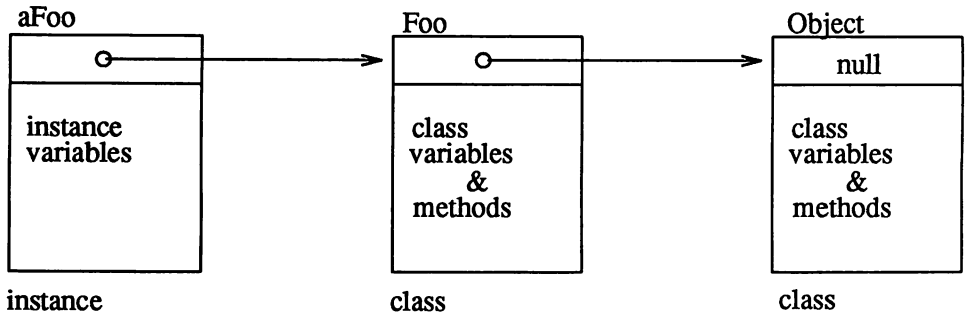


Figure 3 *Relationship between Instances and Classes*

Sending a message to an instance requires packaging the arguments to the method, finding the method in the class chain, invoking the method in the proper context, and possibly returning a result to the sender. If the pseudo-variable *self* is used for the object in sending a message, the search for the method starts at the beginning of the chain, while if *super* is used the search starts

in the superclass.

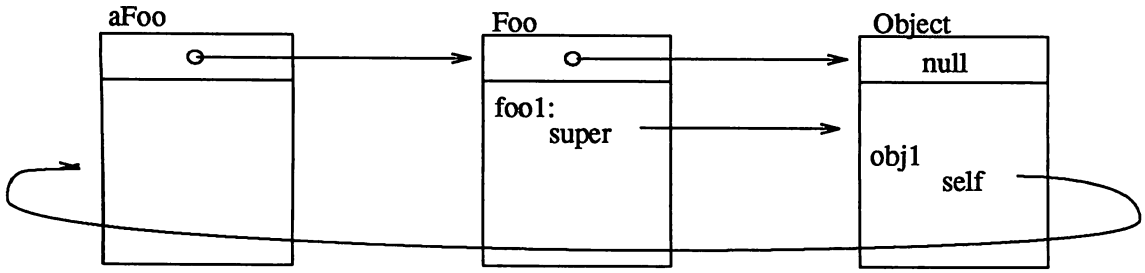


Figure 4 *Self and Super*

PostScript Classes

The PostScript implementation of classes uses dictionaries to represent the classes and instances. Instances contain all the instance variables of all their superclasses. Classes contain their methods as PostScript procedures. Our current implementation of class is entirely in PostScript and is given in Appendix A.

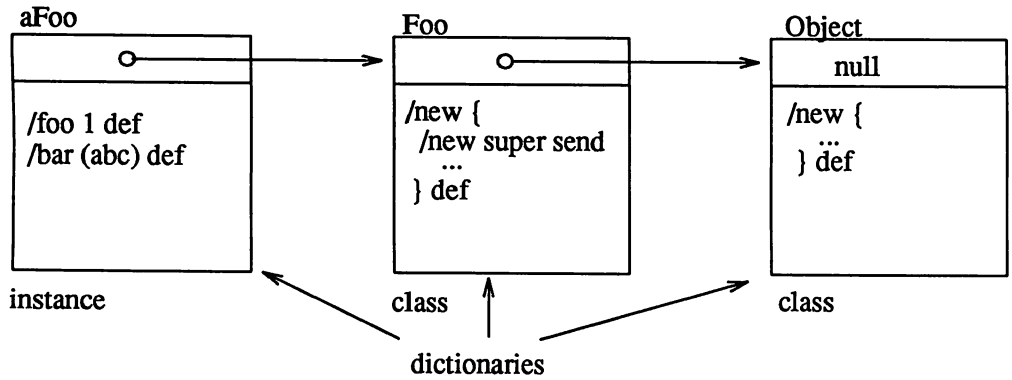


Figure 5 *PostScript use of Dictionaries as Objects*

Classes are built using the *classbegin* . . . *classend* procedures; messages are sent with the *send* primitive:

- **classbegin**: classname superclass instancevariables => –
  Creates an empty class dictionary which is a subclass of *superclass*, and has *instancevariables* associated with each instance of this class. The dictionary is put on the dict stack. *Instancevariables* may be either an array of keywords, in which case they are initialized to *null*, or a dict, in which case they are initialized to the values in the dict.
- **classend**: – => classname dict
  Pops the current dict off the dict stack (put on by *classbegin* and presumably filled in by subsequent *def*s), and turns it into a true class dictionary. This involves compiling the methods and building various data structures common to all classes.
- **send**: <optional args> method object => <optional results>
  Establishes the object's context by putting it and its class hierarchy on the dictionary stack, then executes the method. The method is typically the keyword of a method in the class of the object, but it can be a arbitrary procedure (see the examples below).

The *send* primitive establishes the context for execution of the method by adding the instance and its superclass chain to the dictionary stack. It then executes the method within this context. The

arguments to the method will be on the operand stack as in typical PostScript procedure calls.

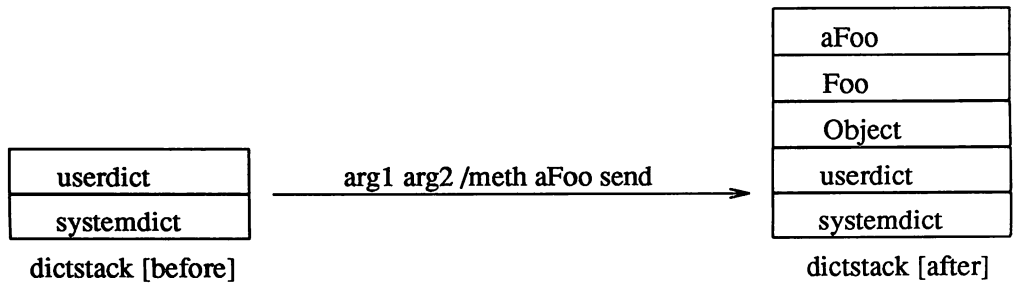


Figure 6 *PostScript use of Dictionaries as Objects*

Example: Class Foo

Here we build a sample class, *Foo*:

```
/Foo Object
dictbegin                    % (initialized) instance variables
    /value 0 def
    /time null def
dictend
classbegin
    /Value [0 1] def         % class variables
    /Time currenttime def

    /new {                   % class methods
        /new super send begin
            /resettime self send
            currentdict
        end
    } def
    /printvars {
        (..we got: Value=%, value=%, Time=%, time=%.\n)
            [Value value Time time] printf
    } def
    /changevalue { % value => - (changes the value of "value")
        /value exch def
    } def
    /resettime { % - => - (sets time to the current time)
        /time currenttime def
    } def
classend def
```

Foo is a subclass of *Object*, discussed above. *Foo* has two instance variables unique to each of its objects; *value*, an arbitrary value associated with the object, and *time*, the time of creation of the object. They are initialized by use of the dict form of specifying instance variables. (The *dictbegin ... dictend* pair are standard utilities which create a dict just the right size for its *def*s.) Similarly, *Foo* has two class variables; *Value*, an arbitrary value associated with the class, and *Time*, the time of creation of the class. Note that *Time* is initialized by calling the PostScript primitive *currenttime*.

Foo has four methods: *new*, *printvars*, *changevalue* and *resettime*. *new* first calls its super class to get a raw instance, which it then initializes by setting the time to the current time. Note the use of *begin ... currentdict end*. This is a common cliché. Also note the use of both self and super: we ask our superclass to make a new instance of itself and initialize it, then ask self to reset our time. *printvars* is used to print the instance and class variables of the object; note how this uses

another standard utility, *printf*. *changevalue* is a method which takes a single argument and assigns it to the instance variable *value*. Finally, *resettime* sets the instance variable *time* to the current time.

Using Class Foo

Let's look at some uses of *Foo*: Here we create a new instance, *foo* of *Foo*. We then print out its initial values, shown by the line starting with "`..we got`".

```
/foo /new Foo send def
/printvars foo send
..we got: Value=array[2], Value=0, Time=1.728, time=4.042.
```

Now we are going to change the value of *foo*'s instance variable *value*. Note that *value* initially was an integer, and we are changing it to a string – an example of PostScript "polymorphism".

```
(Value) /changevalue foo send
/printvars foo send
..we got: Value=array[2], value=0, Time=11.95, time=12.25.
```

Now we do an odd thing, we simply send an executable array (a procedure) to *foo*. The effect of doing this is to execute the procedure within the context of *foo*. (This is somewhat unfair, like cutting paper in Origami, but nicely illustrates the flavor of our combination of PostScript language features and our class extensions.) The procedure we're sending to *foo* is *{Value changevalue}* which assigns *Value*, the class variable, to *value*, the instance variable.

```
{Value changevalue} foo send
/printvars foo send
..we got: Value=array[2], value=array[2], Time=11.95, time=12.25.
```

The above example did not go through method compilation, thus "self" and "super" could not be used. Let's send an executable procedure to *foo* to change *value* to the current value of *Time*, but this time using the more orthodox *doit* method which *does* go through method compilation. The argument to *doit* first sends the message *Time* to itself, which simply returns the value of the *Time* class variable. This then gets sent to *changevalue*. The result is:

```
{/Time self send /changevalue self send} /doit foo send
/printvars foo send
..we got: Value=array[2], value=11.95, Time=11.95, time=12.25.
```

Next we ask *foo* to reset its time value by using the method *resettime*:

```
/resettime foo send
/printvars foo send
..we got: Value=array[2], value=11.95, Time=11.95, time=13.16.
```

Now let's change the class variable *Value* of class *Foo* by use of the *set* method. Because we are not using any of the context of the class, we can send a simple array:

```
[/Value null] /set Foo send
/printvars foo send
..we got: Value=null, value=11.95, Time=11.95, time=13.16.
```

Here we set *Value*, but using the class variable *Time*. This requires deferring the evaluation until within the context of the class, thus use of an executable array:

```
{/Value Time} /set Foo send
/printvars foo send
..we got: Value=11.95, value=11.95, Time=11.95, time=13.16.
```

As a final example, see Appendix B for the PostScript version of the self and super tests on page 62 to 65 in [3]. We used this to convince ourselves that we were implementing self and super correctly!

Class Items

The first real use of classes in NeWS was the class Item. Items are simple, graphical input controls, such as buttons and check-boxes. The Item hierarchy consists of the base class Item, the subclass LabeledItem, and several practical subclasses of LabeledItem. Both Item and LabeledItem provide no usable instances themselves. Rather they are abstract superclasses used to provide a common link for subclasses.

Class Item has these major components:

- A canvas used to depict the item and to be the target of the item's input.
- A set of procedures for painting the canvas and handling activation and tracking events.
- A current value and a procedure which notifies the client when that value changes due to action of the tracking procedures.
- Methods for creating, moving and painting the item, and for returning the item's location and bounding box.

The definition of the class Item is given in Appendix C. Rather than discuss the class in detail, we look at two sample implementations: SampleToggle and SampleSlider, and a simple program that uses them both.

SampleToggle and SampleSlider

SampleToggle provides tracking by implementing the client's Down, Up, Enter, and Exit procedures. The ItemValue is treated as a boolean, with *true* meaning *on*. Down and Enter simply assign `not ItemInitialValue` to ItemValue, while Exit resets it to ItemInitialValue. Up simply calls the notify procedure if the state has changed. SampleToggle adds no instance or class variables.

Here are two toggles, one on and one off, and the implementation of the class SampleToggle:

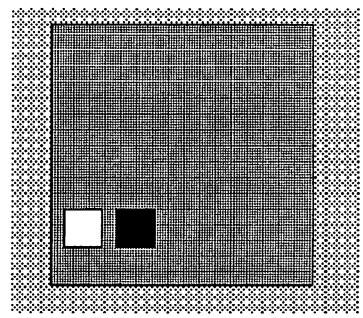


Figure 7 *Two Instances of Class SampleToggle*

```
/SampleToggle Item []
classbegin
    /new { % initialvalue notifyproc parent width height => item
        /new super send begin
            /NotifyUser exch cvx def
            /ItemValue exch def
            currentdict
        end
    } def

    /PaintItem {
        ItemValue
            {0 fillcanvas}
            {1 fillcanvas 0 strokecanvas} ifelse
    } def
    /ClientDown {ItemInitialValue not SetToggleValue} def
    /ClientUp {ItemValue ItemInitialValue ne {NotifyUser} if} def
    /ClientEnter {ClientDown} def
    /ClientExit {ItemInitialValue SetToggleValue} def

    /SetToggleValue { % value => - (set value & paint toggle)
        /ItemValue exch store
        /paint self send
    } def
classend def
```

SampleSlider provides tracking by implementing the client's Down, Up, and Drag procedures. The Down and Drag procedure are identical, simply projecting the current x coordinate of the mouse onto the slider.

Here is a slider and its implementation:

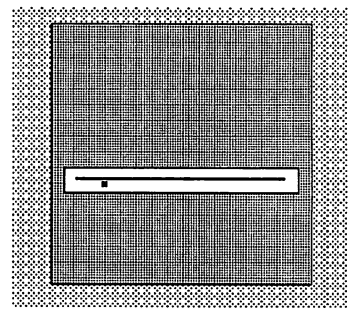


Figure 8 *An Instance of Class SampleSlider*

```
/SampleSlider Item [/SliderX /SliderY /SliderWidth /SliderHeight]
classbegin
    /new { % initialvalue notifyproc parent width height => item
        /new super send begin
            /NotifyUser exch cvx def /ItemValue exch def
            /SliderX            ItemHeight 2 div 1 sub def
            /SliderY            ItemHeight 2 div def
            /SliderWidth        ItemWidth ItemHeight sub def
            /SliderHeight       2 def
            currentdict
        end
    } def
    /PaintItem {
        ItemCanvas setcanvas 1 fillcanvas 0 strokecanvas
        SliderX SliderY SliderWidth SliderHeight rectpath fill
        ItemValue 0 PaintSliderValue
    } def
    /ClientDown {
        SetSliderValue
        ItemValue ItemPaintedValue ne {
            ItemPaintedValue 1 PaintSliderValue
            ItemValue        0 PaintSliderValue
        } if
    } def
    /ClientUp {ItemValue ItemInitialValue ne {NotifyUser} if} def
    /ClientDrag {ClientDown} def
    /PaintSliderValue {  % value gray => -
        setgray
        SliderX add SliderY 5 sub 4 4 rectpath fill
        /ItemPaintedValue ItemValue store
    } def
    /SetSliderValue {
        /ItemValue
            CurrentEvent geteventlocation pop SliderX sub
            0 max SliderWidth min store
    } def
classend def
```

Below is a test program that uses these samples. The notify procedure prints the value of the item using the printf utility. We start by building a canvas and painting it. Then we make two items, a button and a slider, putting them in a dictionary called *items*. We then paint them and fork an activation event manager.

Here's what the test looks like, and its implementation:

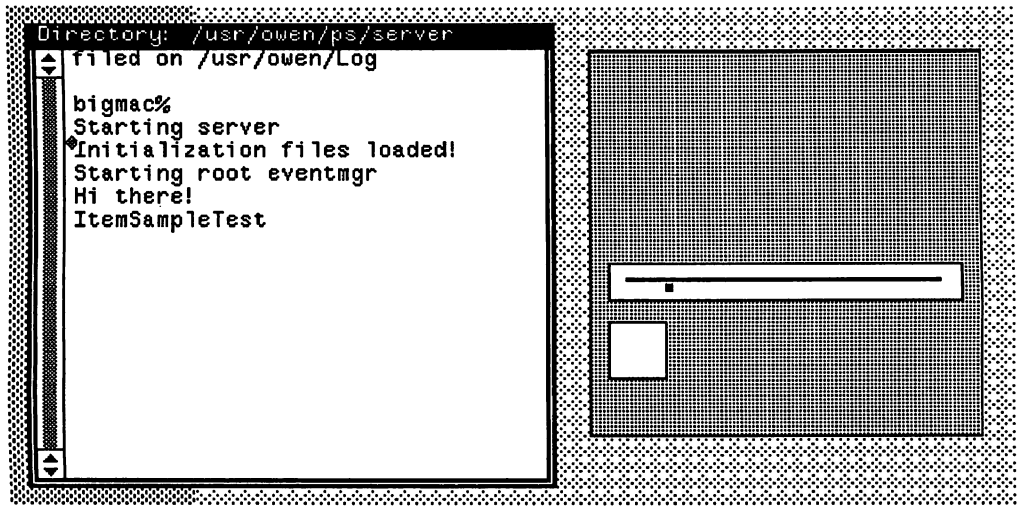


Figure 9 *The Sample Test Program*

```
/ItemSampleTest {
    /notify {ItemValue (ItemValue: % \n) printf} def
    /itembackground .75 def
    /can framebuffer 200 200 createcanvas def
    can setcanvas 200 100 movecanvas currentcanvas mapcanvas
    itembackground fillcanvas 0 strokecanvas

    /items 10 dict dup begin
        /sampletoggle
            false /notify can 30 30 /new SampleToggle send def
            10 30 /move sampletoggle send
        /sampleslider
            20 /notify can 180 20 /new SampleSlider send def
            10 70 /move sampleslider send
    end def
    items paintitems /p items forkitems def
} def
```

After pushing the toggle and sliding the slider, we have:

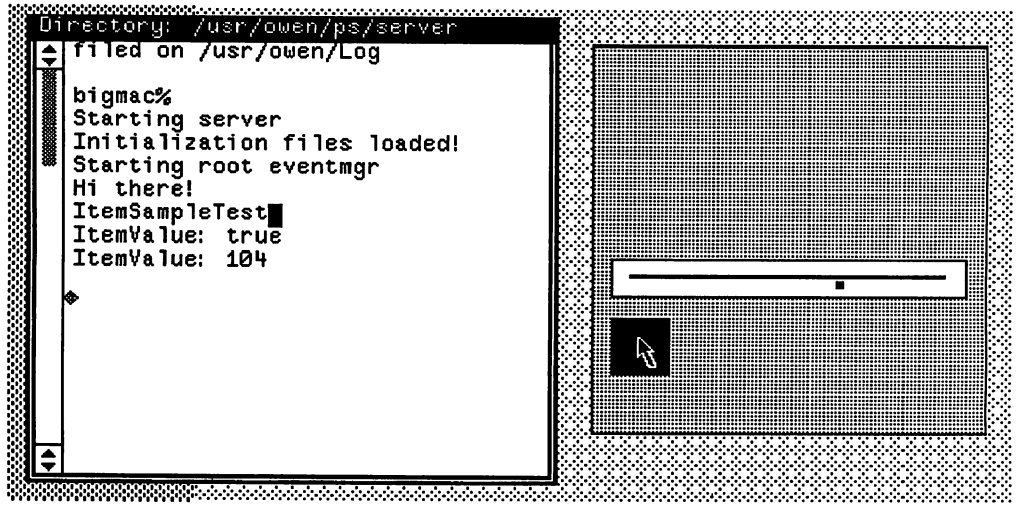


Figure 10 *Use of The Sample Test Program*

What to notice here is the simplicity of the code, emphisizing the power, both of the program and sample items, and of the NeWS programming environment. The implementation of the items and test program in the interpretive NeWS environment took *very* little time.

Class LabeledItem

Most items are more elaborate than the preceeding examples. Class LabeledItem implements a more common item; one that has:

- A polymorphic label-object pair, either of which may be a string, an icon, or a general PostScript procedure.
- A "round rectangle" frame enclosing the item.
- Simple layout rules for automatic positioning of the label and object. The object positon may be to the Right, Left, Top, or Bottom of the label.

The current subclasses of class LabledItem are:

- **ButtonItem**: provides a simple activation/confirmation item
- **CycleItem**: provides check boxes and choices
- **SliderItem**: provides a continuous range of values
- **TextItem**: provides a type-in area
- **MessageItem**: provides an output area
- **ArrayItem**: provides an array of choices

The (abbreviated) class definition is given in Appendix D.

This window contains one of each of these items:

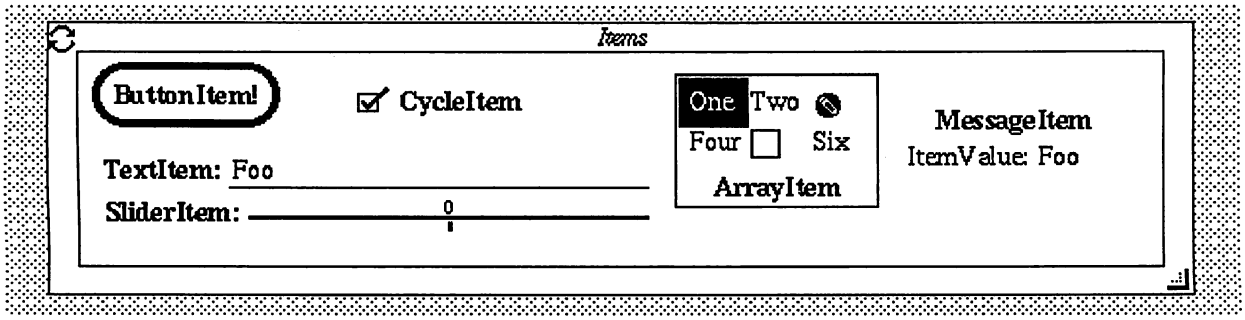


Figure 11 *Subclasses of LabeledItem*

All four object positions are visible in the figure: The text and slider items use */Right*, the cycle item uses */Left*, the message item uses */Top*, and the array item uses */Bottom*. The complete class definition for one of these, CycleItem, is given in Appendix E.

Summary

Use of a Smalltalk-like class mechanism proves to have several advantages for NeWS:

- Classes are a well documented standard discussed in several easily obtainable books.
- Classes formalize the flexibility needed by the NeWS community.
- There are at least two well-documented class hierarchies for application writing: Smalltalk itself, and MacApp, Apple's "extensible application."
- Classes are easily and naturally implemented in PostScript.
- Classes offer rapid prototyping and high productivity.

Our initial implementation of classes was done in PostScript itself, with no extensions required to the interpreter. We then implemented a reasonably complex class hierarchy for standard user interaction items. PostScript's polymorphism proved quite useful for yielding different results for different items. For example, TextItems return PostScript strings, SliderItems return an integer, and ArrayItems return a 2 element array index. We feel our initial use of classes has been successful. We plan to use classes for the package level of NeWS.

References

1. Kurt J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden Book Company, 1986.
2. *Byte Magazine*, special issue on *Object-Oriented Languages*, August, 1986, McGraw-Hill Inc.
3. Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, May, 1983.
4. *NeWS Preliminary Technical Overview*, Sun Microsystems, October 2, 1986.
5. Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison Wesley, July, 1985.

Appendix A: Complete listing of PostScript class implementation

```
%
%  Objects 'n stuff.
%

/ObjectTemplate dictbegin  % All objects have these fields:
    /parentDict      null def  % link to my parent dict; stops at null.
    /parentDictArray null def  % complete parent chain to Object!
dictend def
/ClassTemplate dictbegin  % Class objects have these fields in addition:
    /instanceVars     null def % this class' additional inst vars
    /instanceVarDict  null def % this class' total inst vars
    /instanceVarExtra 10 def   % extra space for class var over-rides
    /className        null def % name of the class (as a keyword)
    /subClasses       nullarray def  % subclass list (for browsing)
dictend def

% Create a sub-class of the given class.
%    The instancevariables may be either an array or a dict.
%    The advantage of using a dict is that the variables will be
%    pre-initialized to a value you chose, rather than "null".
/classbegin {   % classname superclass insvars => - (newclass on dict stack)
dictbegin
    ObjectTemplate {def} forall
    ClassTemplate {def} forall

    /instanceVars exch def
    /parentDict exch def
    /className exch def
} def
/classend { %  - => classname newclass
    currentdict {
        dup xcheck {parentDict methodcompile def} {pop pop} ifelse
    } forall

    /instanceVarDict instanceVars def
    /parentDictArray [] def

    instanceVarDict type /arraytype eq {
        /instanceVarDict instanceVarDict length dict dup begin
            instanceVarDict {null def} forall
        end def
    } if

    parentDict null ne {
        parentDict /subClasses 2 copy get [className] append put
        /instanceVarDict
            parentDict /instanceVarDict get instanceVarDict append def
        /parentDictArray
            parentDict /parentDictArray get [parentDict] append def
    } if

    className
dictend
} def
% Crack open the methods and fix for "super send" and "self send"
/methodcompile { % method parentdict => newmethod
10 dict begin
    /superpending false def
    /selfpending false def
```

```
    /parentDict exch def
    [ exch {
        dup /send eq superpending selfpending or and {
            pop pop
            superpending
                {parentDict /className get cvx /supersend cvx}
                {cvx} ifelse
        } if
        dup type /arraytype eq {parentDict methodcompile} if

        dup /super eq /superpending exch def
        dup /self eq /selfpending exch def
    } forall
    ] cvx
end
} def

% Generic Smalltalk-ish Primitives.
% Send a message to an object.
/send { % <args> message object => <results>
    dup /parentDictArray get {begin} forall
    begin
        cvx exec
    parentDictArray length 1 add {end} repeat
} def
% Send a message to super without poping myself.
/supersend { % <args> keywordmessage superclass => <results>
    exch { 2 copy known {exit} {exch /parentDict get exch} ifelse } loop
    get exec
} def
% Put me on the operand stack.
/self {/parentDict where pop} def

% Your basic object!
/Object null [] classbegin
    /new { % class  =>  instance
        ObjectTemplate length instanceVarDict length instanceVarExtra
        add add dict dup begin
            instanceVarDict {def} forall
            ObjectTemplate {def} forall
        end
        dup /parentDict currentdict put
        dup /parentDictArray parentDictArray [currentdict] append put
    } def
    /doit { % proc ins  =>  - (compile & execute the proc)
        parentDict /parentDict get methodcompile exec
    } def

    /set { % {/key value ...}  =>  -  stores the values in the object
        mark exch cvx exec
        counttomark 2 div {def} repeat pop % store??
    } def
classend def
```

Appendix B: Smalltalk [3] page 62 example in PostScript

This is a PostScript implementation of the self and super tests given in [3] on pages 62 - 65.

```
/smalltalkpage62 {
    /One Object [] classbegin
        /test {1} def
        /result1 {/test self send} def
    classend def

    /Two One [] classbegin
        /test {2} def
    classend def

    /ex1 /new One send def
    /ex2 /new Two send def

    /test ex1 send =
    /result1 ex1 send =
    /test ex2 send =
    /result1 ex2 send =

    /Three Two [] classbegin
        /result2 {/result1 self send} def
        /result3 {/test super send} def
    classend def
    /Four Three [] classbegin
        /test {4} def
    classend def

    /ex3 /new Three send def
    /ex4 /new Four send def

    /test ex3 send =
    /result1 ex4 send =
    /result2 ex3 send =
    /result2 ex4 send =
    /result3 ex3 send =
    /result3 ex4 send =
} def
```

Results:

```
smalltalkpage62
1 1 2 2 2 4 2 4 2 2
```

Appendix C: Class Item

```
/Item Object [
% instance variables
    /ItemWidth        % item's width,
    /ItemHeight       % ..& height,
    /ItemParent       % ..& parent canvas (from "new")
    /ItemCanvas       % the canvas we created for the item
    /ItemValue        % the canvas' current value
    /ItemInitialValue % the value it started out with
    /ItemPaintedValue % the value it currently shows
    /StartInterest    % the interest which activates the item
    /ItemInterests    % interests used to track item
    /ItemEventMgr     % ..the tracking process
    /NotifyUser       % the user's notify proc
] classbegin
% default variables
    /ItemFont           DefaultFont def    % the item's font
    /ItemTextColor      0 0 0 rgbcolor def % ..& text color
    /ItemBorderColor    ItemTextColor def  % ..& border color
    /ItemFillColor      1 1 1 rgbcolor def % ..& background color
% class variables; mainly the std client procs
    /PaintItem   nullproc def  % the core of the /paint method
    /ClientDown  nullproc def  % procedures installed in
    /ClientDrag  nullproc def  %   the activated (tracking)
    /ClientEnter nullproc def  %   process
    /ClientExit  nullproc def
    /ClientKeys  nullproc def
    /ClientUp    nullproc def
    /StopOnUp?   true def      % deactivate on up event?
% methods
    /new               % parentcanvas width height => instance
    /makecanvas        % - => -
    /makeinterests     % - => -
    /move              % x y => -  (Moves item to x y)
    /moveinteractive   % items backgroundcolor => -
                       % (interactively moves the item)
    /paint             % - => -  ([Re]paints item)
    /location          % - => x y
    /bbox              % - => x y w h
classend def
```

Appendix D: Class LabeledItem

```
/LabeledItem Item
dictbegin
% instance variables
    /ItemObject     nullstring def % The item's "object"
    /ObjectX        0 def          % and bounding rect:
    /ObjectY        0 def
    /ObjectWidth    0 def
    /ObjectHeight   0 def
    /ItemLabel      nullstring def % The item's "label"
    /LabelX         0 def          % and bounding rect:
    /LabelY         0 def
    /LabelWidth     0 def
    /LabelHeight    0 def
    /ItemBorder     2 def    % Extra space around the item
    /ObjectLoc      null def % Label-Object position
    /ItemGap        5 def    % Distance between object & label
    /ItemFrame      0 def    % Draw frame if not zero
    /ItemRadius     0 def    % Radius of frame
dictend
classbegin
% default variables
    /ItemLabelFont  Item /ItemFont get def
% class variable: over-ride of PaintItem
    /PaintItem       % - => -
% methods: over-ride new
    /new  % label obj loc notify parent width height => instance
% utilities used to manipulate label-object pair
    /LabelSize       % - => width height
    /ShowLabel       % - => -
    /ShowObject      % - => -
    /EraseObject     % - => -
    /AdjustItemSize  % - => -
    /CalcObj&LabelXY % - => -
classend def
```

Appendix E: Class CycleItem

```
/CycleItem LabeledItem
dictbegin
    /ItemValue        0 def
    /EraseToUpdate    true def % erase when switching state
    /Cycle            nullarray def
dictend
classbegin
    /new { % label array loc notify parent width height => instance
        /new super send begin
            /Cycle ItemObject def
            currentdict
        end
    } def
    /makecanvas {
        BindCycleObject
    % calculate Label & Object Height & Width:
        Cycle { % calculate bbox for all the cycle objects
            ItemFont ThingSize
            /ObjectHeight exch ObjectHeight max def
            /ObjectWidth exch ObjectWidth max def
        } forall
        LabelSize /LabelHeight exch def /LabelWidth exch def

        AdjustItemSize
        CalcObj&LabelXY

        /ItemCanvas ItemParent ItemWidth ItemHeight createcanvas def
    } def
    /PaintItem {/PaintItem super send false PaintCycle} def
    /SetCycleValue { % Bump? => - (Set ItemValue to initial or bumped value)
        /ItemValue ItemInitialValue 3 -1 roll
            {1 add Cycle length mod} if store
        ItemValue ItemPaintedValue ne {
            true PaintCycle
            /ItemPaintedValue ItemValue store
        } if
    } def
    /BindCycleObject {/ItemObject Cycle ItemValue get store} def
    /ClientDown {true SetCycleValue} def
    /ClientUp {ItemValue ItemInitialValue ne {NotifyUser} if StopItem} def
    /ClientEnter {true SetCycleValue} def
    /ClientExit {false SetCycleValue} def

    /PaintCycle { % updating? => -
        EraseToUpdate and {EraseObject} if
        BindCycleObject ShowObject
    } def
classend def
```

COMPUTER ASSISTED COLOR CONVERSION sm

David M. Geshwind

Digital Video Systems
111 Fourth Avenue
New York, NY 10003
/212/777-2898

Latent Image Development Corp.
Two Lincoln Square
New York, NY 10023
/212/787-1275

INTRODUCTION:

The process of adding computerized color to black and white motion pictures and television programs has been hailed as "an ingenious technological breakthrough" and derided as "cultural butchery". Businessmen see it as a means of breathing new life into unsaleable media properties while the original filmmakers (who do not share in the new income) see it as "vandlism" of classic works of art.

While it is a matter of opinion as to whether colorization is a good idea in and of itself, it is clear that bad colorization is not. The current products available for broadcast and home video have been almost uniformly washed out, over-simple and unsubtle. Film critic Gene Siskel observes,

> "There's a certain timidity about the colors used, lot's of pastels, as in ... Topper. The coloring of Yankee Doodle Dandy is a visual mess. In some scenes everything is awash with blue except the skin tones, which are the same for every actor. So much for the actors' individuality ... The coloring is even more laughable in the classic It's A Wonderful Life ... And the classic gaffs; look at Frank Sinatra in Suddenly, 'Ol' Blue Eyes' is back ... as 'Ol' Brown Eyes'."

His cohort Roger Ebert continues,

> "They tend to pick light blues, light greens, pinks, violets, yellows ... in the 30s Cagney looked like he was going out to commit murder, colorized he looks like he's going out to play golf."

These defects are not inherent in our colorization process. Nor does the work now being done at commercial colorization facilities have to be so poor. The low quality of current colorization services is a matter of economics and poor design, not technological limitations.

To understand the trade-offs one must realize that the popular notion - that an 'artist' colors only one frame of a scene and the 'computer' colors all the rest - is a gross exageration. Artificial intelligence and pattern recognition has had limited success in being able to extract known objects (e.g. armored tanks) in restricted situations (e.g. sitting out in the open). Tracking and extracting multiple, unfamiliar, moving objects, that mutually overlap, in contexts that change every few seconds with each camera shot, is beyond the current state-of-the-art. This research may one day result in an affordable, fully automated colorization technology but, for the forseeable future, continual human judgement and interaction is still a necessary component of computerized colorization systems.

Although the process is highly computer assisted, and much more efficient than hand painting each frame, it is still very labor intensive. The colorization operator must outline each separately colored object in a very large proportion of the film's frames. And, since labor is the most expensive element of the process, the fewer distinct objects and the fewer colors in a scene, the less colorization costs. Mimicing the visual complexity inherent in a 'real' color scene is very expensive.

The choice of washed-out pastels is made for a different reason. With thousands of frames to paint, colorizers can afford to spend little time getting the details correct on each one. And the computer process that 'colors' the interviening frames is not always accurate. The greater the movement in the scene the more these problems are apparent. Siskel notes,

> "Coloring also stumbles when the actors move. Look at the color bleeding in the dance scenes of Yankee Doodle Dandy."

The inaccuracies in the placement of colors are much less noticable if the colors are less saturated. In some converted films the colors are so washed-out they can barely be seen - but then neither can the mistakes. However, in the current market it is not visual quality which is important but the ability to reclassify a film as color. Says Ebert,

> "It's a story about money, not about art".

NEW APPROACH:

We have developed a newly patented technology for film colorization that has a number of advantages over current techniques. The underlying principle of our approach is to maintain the new color information as a separate component until very late in the process. This allows us to apply various data compression techniques when creating and processing the color component, without degrading the original black and white image information in any way.

The human visual system is less sensitive to color detail than to black and white detail. Detection of edges, perception of shading and other detail functions depend primarily upon the luminance rather than the chrominance aspect of a visual scene. Thus, when

a low information density color overlay signal is combined with a high density luminance signal the result is percieved as a high density full color image.

One application allows for the production of colorized product in virtually any format with only moderate processing requirements. Current systems transfer the black and white film to videotape as a first step and thereby immediately discard as much as 90% or more of the original information. A color signal is generated, also at video resolution, and the two are combined electronically. Therefore, color converion production is limited to videotape product. Scaling the entire system up to high-definition video is not now economically practical and would still not produce results suitable for theatrical projection. However, our approach allows us to create a color overlay film at moderate resolution and optically combine it with the original black and white film as the finaly step. In this way, a composite film can be produced economically, that will be perceived as both full color and high definition.

This same technique can be applied to a computer assisted system that will permit economical restoration of faded or damaged color film.

When colorizing for video product we can reduce the data density of the color signal even further, permitting very economical production on relatively low-tech systems. By applying these techniques, the amount of computer processing time, human interaction, storage, and overall, the need for highly complex high-speed systems, can be greatly reduced. All these factors result in making the process much more economical.

This reduction in operating costs makes it economically feasible to take the time needed to colorize with more care and subtlety and with more sophisticated design.

DATA COMPRESSION:

We can generate a low information density signal by compression of the digital representation of the color specification in any combination of three ways:

Spatial Compression: the color information is not specified at all points on a given frame. This may be as simple as working with uniformly larger pixels, or may involve creation of a low detail representation, such as a polygonal outline of objects.

Temporal Compression: the color information is not specified for every frame. The same color frame information may simply be repeated for a sequence of several black and white frames, or shape interpolation (inbetweening) may be applied to the color areas.

Color Choice Compression: the amount of information used to specify color at each pixel may be reduced uniformly throughout by restricting the 'color space' within which the image exists. Alternately, a single color specification may be applied to a large group of pixels, such as within a polygonal area.

Visable anomalies may result from compression or undersampling; spatial defects include 'the jaggies'; temporal defects include jerkyness, doubling or strobing; color choice defects include contouring or 'flatness'. The perception of these defects can be reduced or eliminated by various antialiasing techniques; inbetweening or cross-dissolving in the temporal domain; digital or analog low-pass filtering or optical defocusing for spatial and contouring defects. Also, to counter a uniform 'flat' look to objects, additional variation to the color information may be added, as detail, at a later stage in the process.

TYPICAL SYSTEM OPERATION

What follows is a practical comparison of the requirements for coloring a one second sequence of film using both straight-foward digital image processing and our approach. We will assume that the scene is a non-action scene where the motion of objects is moderate, as is typical with much motion picture material.

By straight-foward digital image processing, if theatrical film output is desired, scanning rates would have to be in the range of 1500 lines per frame and data depth of about 24-bits per pixel to accomodate full color variation. This results in:

1500 lines x 1500 lines = 2.25 million pixels/frame

x 3 bytes/pixel = 6.75 million bytes/frame

x 24 frames/second = 162 million bytes/second.

Assuming that ten objects are to be separately colored in each frame, and that an artist can paint these ten objects in two minutes, 48 minutes of operator time will be required to paint these 24 frames.

Using our approach, we will be leaving the high density luminance information in the film domain. We will therefore be creating chrominance information at a more moderate resolution of 500 lines/frame. Color specificaiton will be limited to 8-bits per pixel which will allow for many separately tagged objects with some variation within objects (e.g. 32 objects with up to eight hue variations within each object).

Operator outlining of objects will be limited to one out of six frames. The intervening even frames will be created by inbetweening the positions and shapes of the objects; all odd frames will be created by cross-disolving the color information of the two adjecent even frames during optical printing.

Storage requirements are:

500 lines x 500 lines = .25 million pixels/frame

x 1 bytes/pixel = .25 million bytes/frame

x 12 (even) frames/second = 3 million bytes/second.

The operator requires 8 minutes to 'paint' four frames. We will assume that it will require an additional 2 minutes to check the two interpolated frames and hand correct for one or two objects that moved too erratically for inbetweening to work correctly.

By applying these techniques a 54-fold decrease in information creation and storage requirements has been achieved and a savings of almost 80% of the human labor.

Variations on the process may provide additional efficiencies of operation or add subtleties to the composite image. For example, the availablity of high-speed polygon rendering hardware makes it possible to create, store and process only the object outline data, on the order of a kilobyte per frame, rather than the full-density, pixel oriented, color information, which is 100s or 1000s of kilobytes per frame. The outline data is then fed to one of these 'rendering engines' for at or near real-time display and recording.

In another implementation, the luminance information is scanned in at the same low information density as the chroma information. Although this version of the luminance information will ultimately be discarded, to be replaced by the original high density version, it is used to modulate the chroma information. The hue of individual image areas may be varied on a pixel by pixel basis to avoid a 'flat' unrealistic look. For example, rather than color a person's face a single uniform pink hue, the highlights might be made more yellow and the shadows slightly blue to give the effect of sunlight.

IN CONCLUSION

Improved colorization technology will increase production quality and satisfy many of the critics of the process as it is currently practiced. However, purists argue that any colorization process is a desecration of pristine classics.

It should be noted that older films, particularly when shown on television, are hardly maintained as originally intended. Edited to fit time slots, interupted by commercials, displayed on a small, high contrast CRT, bandlimited to a fuzzy 4 MHz; home viewing is vastly different than the theater experience. But it is tollerated as a way to secure wide and continued distribution for a large and varied catalog of films.

There are many other films that are equally worthwhile but which are non-existant, in practice, because it is not economically viabel to broadcast black and white, and because large sections of the audience shun black and white programs even when they are shown. Colorization makes these films available again and, for a significant portion of the population, increases their enjoyment of these programs.

List of Attendees

Sara Abatiell
IBM, Inc.
1510 Page Mill Road
MS 35A
Palo Alto, CA 94304
(415)855-3698

Brent Auernheimer
CA State University-Fresno
Department of Computer Science
Fresno, CA 93740-0109
(209)294-4373
ucbvax!ucsbcsl!csufres!brent

Michael J. Bailey
Megatek Corporation
9645 Scranton Rd.
San Diego, CA 92121
(619)445-5590

William Beblo
Bell Communications Research
290 West Mt. Pleasant Avenue
Room 1D-148
Livingston, NJ 07039
(201)740-4421
...ulysses!gamma!wb

Richard A. Becker
AT&T Bell Laboratories
600 Mountain Avenue
Room 2C259
Murray Hill, NJ 07974
(201)582-5512
research!rab

Larry Bickford
Qubix Graphic Systems
1255 Parkmoor Avenue
San Jose, CA 95126
(408)292-4000
decwrl!qubix!lab

Doug Blewett
AT&T Bell Laboratories
600 Mountain Ave.
Room 2C-548
Murray Hill, NJ 07974
(201)582-6496
research!blewett

Reidar Bornholdt
Columbia University
630 West 168th Street
New York, NY 10032
(212)705-3411
{harpo,cmcl2}!cucard!reidar

David Boyle
Compugraphic Corporation
200 Ballardvale St.
MS 200-35S
Wilmington, MA 01887
(617)658-5600

Charles Brauer
Fairchild Research Center
4001 Miranda
Palo Alto, CA 94304
(415)858-4572

Peter Broadwell
Silicon Graphics
2011 Stierlin Road
Mountain View, CA 94043
(415)960-1980
sgi!peter

Jeff Bulf
Qubix Graphic Systems
1255 Parkmoor Avenue
San Jose, CA 95126
(408)292-4000
decwrl!qubix!jeff

Mark Callow
Silicon Graphics, Inc.
2011 Stierlin Road
Mountain View, CA 94043
(415)960-1980
sgi!msc

George Collier
Bell Communications Research
435 South Street
Room MRE 2Q373
Morristown, NJ 07960
(201)829-4762
bellcore!collier

Chris Crampton
Sci & Eng Research Council
Rutherford Appleton Laboratory
Informatics Division
Chilton, Bidcot Oxf, England
44 23 52 19 00

Steve Cunningham
CA State University-Stanislaw
Computer Science
Turlock, CA 95380
(209)607-3176
...!lll-crg!csustan!rsc

Owen Densmore
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
(415)691-7474
odensmore@sun

Judith DesHarnais
USENIX Association
P.O. Box 385
Sunset Beach, CA 90742
(213)592-3243
usenix!judy

Edward Donofrio
Bell Communications Research
435 South Street
Room 2E-341
Morristown, NJ 07960
(201)829-4409
bellcore!ejd

Tom Duff
AT&T Bell Laboratories
600 Mountain Avenue
Room 2C-425
Murray Hill, NJ 07974
(201)582-6485
research!td

Bruce Ellis
AT&T Bell Laboratories
600 Mountain Avenue
Room 2C-420
Murray Hill, NJ 07974
(201)582-4222
research!brucee

David Elrod
Sun Microsystems
2550 Garcia Avenue
Mountain View, CA 94043
(415)691-6179
ucbvax!sun!dhelrod

Adrian Freed
Consultant
22 Sirard Lane
San Rafael, CA 94901
(415)485-9790
pixar!tdv!adrian

David M. Geshwind
Latent Image Development Corp.
Two Lincoln Square
New York, NY 10023
(212)787-1275

Mr. A. Gokhale
Godrej & Boyle Mfg. Co.
Godrej Bhavan
Home Street
Bombay 400001, India
91 22 204-8371

Dr. Julian Gomez
NASA-RIACS
NASA Ames Research Center
MS 230-5
Moffett Field, CA 94035
(415)694-6141
julian@riacs.edu

David Gorgen
Apollo Computer
330 Billerica Road
Chelmsford, MA 01824
(617)256-6600
decvax!wanginst!apollo!dpg

Chuck Grant
Lawrence Livermore Natl Lab
P.O. Box 5504
L-156
Livermore, CA 94550
(415)422-7278
grant@lll-icdc.arpa

Dany Guindi
Georgia Inst. of Technology
Software Eng. Research Center
Information & Computer Science
Atlanta, GA 30332
(404)894-7512
dany@gatech.edu

Paul Haeberli
Silicon Graphics
2011 Stierlin Road
Mountain View, CA 94043
(415)960-1980

Tony Hasegawa
GE/NASA Ames
M/S 223-2
Moffett Field, CA 94035
(415)694-6286
ucbvax!ames-prandtl.arpa

Michael Hawley
MIT
Media Lab, E15-493
20 Ames Street
Cambridge, MA 02139
(617)253-0621
media-lab.mit.edu!mike

Conrad Huang
Univ. of CA - San Francisco
Computer Graphics Laboratory
School of Pharmacy
San Francisco, CA 94143-0446
(415)476-9156
conrad@cgl.ucsf.edu

Carol Hunter
Lawrence Livermore Natl Lab
P.O. Box 808
L301
Livermore, CA 94550
(415)422-1657

William Johnston
Lawrence Berkeley Laboratory
1 Cyclotron Road
Building 50B/3238
Berkeley, CA 94720
(415)486-5014
ucbvax!lbl-csam.arpa!johnston

Lou Katz
Metron Computerware Limited
3317 Brunell
Oakland, CA 94602
(415)530-1430
ucbvax!lou

Ashok Khosla
Ampex Corporation
401 Broadway
Redwood City, CA 94063-3199
(415)367-3857
decwrl!turtlevax!avsd!avsdt:ashok

Ken Knowlton
Consultant
1613 New Brunswick Avenue
Sunnyvale, CA 94087
(408)732-5044

Stuart Kreitman
Conversion Technologies
2314 North First Street
San Jose, CA 95134
(408)435-3641

Peter Langston
Bell Communications Research
435 South Street
2D396
Morristown, NJ 07960
(201)829-4332
belcore!psl

Jaron Lanier
VPL Research Inc.
656 Bair Island Road
Suite 304
Redwood City, CA 94063
(415)361-1710

Allen Leinwand
Qubix Graphic Systems
1255 Parkmoor Ave
San Jose, CA 95126
(408)292-4000
decwrl!qubix!allen

Creon Levit
NASA Ames Research Center
M/S 233-1
Moffett Field, CA 94035
(415)694-6410
creon@ames-nas

Steve Lowder
Hewlett Packard Labs
4161 George Ave #4
San Mateo, CA 94403
(415)341-5939

Allen McPherson
Boeing Computer Services
P.O. Box 24346
M/S 6R-51
Seattle, WA 98124
(206)656-5446

Rebecca Meacham
Bell Communications Research
290 W. Mt. Pleasant Ave.
LCC 1B-128
Livingston, NJ 07039
(201)740-4397
bellcore!gamma!crm

Richard Mossman
Pacific Bell
120 Montgomery Street
Room 330
San Francisco, CA 94104
(415)774-8449
{ihnp4,qantel,pyramid,}ptsfa!rkm

Rob Myers
Silicon Graphics
2011 Stierlin Road
Mountain View, CA 94043
(415)960-1980
sgi!rob

Charles Perkins
IBM
T.J. Watson Research Center
P.O. Box 218, Room 03-129
Yorktown Heights, NY 10598
(914)945-3969
philabs!polaris!charliep

John Peterson
Univ. of Utah
Department of Computer Science
M.E.B.
Salt Lake City, UT 84112
(801)581-6678
peterson@utah-cs

Robert Picco
Compugraphic Corporation
200 Ballardvale St.
MS 200-35S
Wilmington, MA 01887
(617)658-5600

John Pittman
Wavefront Technologies
530 East Montecito Street
Santa Barbara, CA 93103
(805)962-8117

Michael Pulliam
G.E. Calma Company
R&D Engineering, M/S C51
525 Sycamore Drive
Milpitas, CA 95035-7489
(408)943-5970
...ucbvax!calma!pulliam

Brian Redman
Bell Communications Research
435 South Street
Room MRE 2E-388
Morristown, NJ 07960
(201)829-2884
bellcore!ber

Joseph Requa
Lawrence Livermore Natl. Lab
P.O. Box 80
L-61
Livermore, CA 94550
(415)422-4036

Scott Ritchie
Sun Microsystems
2550 Garcia Avenue
5-40
Mountain View, CA 94040
(415)691-7434
sritchie@sun

Craig Rodine
AT&T Bell Laboratories
600 Mountain Avenue
Room 5E-102
Murray Hill, NJ 07974
(201)582-6907
ucbvax!research!crr

David Rosenthal
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
(415)691-7569
dshr@sun.comp

Linda Rybak
Department of Defense
9800 Savage Road
Ft. Meade, MD 20755
(301)688-6693

Peter Salus
USENIX Association
P.O. Box 7
El Cerrito, CA 94530
(415)528-8649
usenix!peter

David Scharpf
Boeing Computer Services, Inc.
P.O. Box 24346
M/S 9R-48
Seattle, WA 98124
(206)237-8908

Carlo H. Sequin
Univ. of CA - Berkeley
Department of EECS
Berkeley, CA 94720
(415)642-5103
sequin@berkeley.edu

Bo Sjosten
Graphic Software Systems
P.O. Box 4900
Beaverton, OR 97005
(408)720-8072

Nigel Stephens
Whitechapel Workstations
75 Whitechapel Road
London E1 1DU, England
44-1-377-8680
nigel@wcwvax.co

Johan Strandberg
Apple Computer
10500 N. De Anza Boulevard
Cupertino, CA 95014
(408)973-2749

Ed Tannenbaum
Eddeo Inc.
P.O. Box 92
Crockett, CA 94525
(415)787-1567

Jack A. Test
Alliant Computer Systems Corp.
One Monarch Drive
Littleton, MA 01460
(617)486-1203
decvax!linus!alliant!jat

Spencer W. Thomas
Univ. of Utah
Computer Science Department
Salt Lake City, UT 84112
(801)581-3095
thomas@utah-cs

David Tristram
NASA Ames Research Center
M/S 233-1
Moffett Field, CA 94035
(415)694-6410
dat@ames-nas

James Van Loo
Sun Microsystems
MS5-40
2500 Garcia Avenue
Mountain View, CA 94022
(415)960-1300
jvanloo.sun.com

Blaine Walker
GE/NASA Ames
M/S 223-2
Moffett Field, CA 94035
(415)694-6286
ucbvax!ames-prandtl.arpa

Myron Wish
AT&T Bell Laboratories
600 Mountain Avenue
Room 2C 575
Murray Hill, NJ 07974
(201)582-7630
research!mikey

Jean Wood
Digital Equipment Europe
B.P. 29 Sophia Antipolis
Valbonne, France 06561
33 93 65 50 31
decvax!jean

Michael York
Boeing Computer Services
P.O. Box 24346
M/S 6R-51
Seattle, WA 98124
(206)656-5446
uw-beaver!ssc-vax!voodoo!zombie

David Yost
Grand Software, Inc.
8464 Kirkwood Drive
Los Angeles, CA 90046
(213)650-1089
{attmail|usenix}grand!dyost